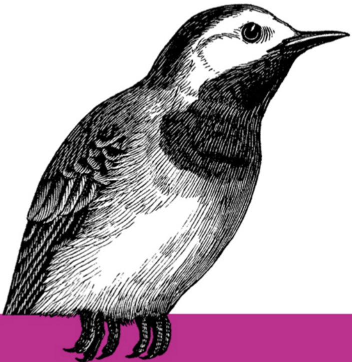
O'REILLY®

图灵程序设计丛书





Git团队协作

掌握Git精髓，解决版本控制、工作流问题，实现高效开发

Git for Teams



[加] Emma Jane Hogbin Westby 著
童仲毅 译

中国工信出版集团
人民邮电出版社
POSTS & TELECOM PRESS

数字版权声明

图灵社区的电子书没有采用专有客户端，您可以在任意设备上，用自己喜欢的浏览器和PDF阅读器进行阅读。

但您购买的电子书仅供您个人使用，未经授权，不得进行传播。

我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。

译者介绍

童仲毅

学生开发者，毕业于复旦大学软件学院。开源应用作者，作品在GitHub上获得上万 Star，被数百万用户使用。对一切未知充满好奇，梦想走遍世界的每一个角落。GitHub ID：geeeeeeeeek。

图灵程序设计丛书

Git团队协作

Git for Teams

[加] Emma Jane Hogbin Westby 著
童仲毅 译

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo
O'Reilly Media, Inc.授权人民邮电出版社出版

人民邮电出版社
北 京

图书在版编目（C I P）数据

Git团队协作 / (加) 艾玛·简·霍格宾·韦斯特比著 ; 童仲毅译. -- 北京 : 人民邮电出版社, 2017.6
(图灵程序设计丛书)
ISBN 978-7-115-45467-6

Ⅰ. ①G… Ⅱ. ①艾… ②童… Ⅲ. ①软件工具—程序设计 Ⅳ. ①TP311.561

中国版本图书馆CIP数据核字(2017)第084097号

内 容 提 要

本书是一本软件团队协作指南，采用以人为本的方式讲解版本控制，强调如何利用 Git 促进团队协作。第一部分介绍如何创建一个优秀的团队、如何构建工作流等。第二部分从实践的角度学习 Git 命令。第三部分介绍如何在 GitHub、Bitbucket 和 GitLab 平台上托管项目。

本书适合软件开发人员和项目管理人员阅读。

◆ 著　[加] Emma Jane Hogbin Westby
译　童仲毅
责任编辑　岳新欣
执行编辑　赵瑞琳
责任印制　彭志环

◆ 人民邮电出版社出版发行　北京市丰台区成寿寺路11号
邮编　100164　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　印刷

◆ 开本：800×1000　1/16
印张：16.75
字数：396千字　2017年 6 月第 1 版
印数：1 – 3 500册　2017年 6 月北京第 1 次印刷
著作权合同登记号　图字：01-2017-0780号

定价：69.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京东工商广字第 8052 号

版权声明

© 2015 by Emma Jane Hogbin Westby.

Simplified Chinese Edition, jointly published by O'Reilly Media, Inc. and Posts & Telecom Press, 2017. Authorized translation of the English edition, 2015 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.

英文原版由 O'Reilly Media, Inc. 出版，2015。

简体中文版由人民邮电出版社出版，2017。英文原版的翻译得到 O'Reilly Media, Inc. 的授权。此简体中文版的出版和销售得到出版权和销售权的所有者——O'Reilly Media, Inc. 的许可。

版权所有，未得书面许可，本书的任何部分和全部不得以任何形式重制。

O’Reilly Media, Inc.介绍

O’Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O’Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O’Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O’Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 *Make* 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O’Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O’Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过图书出版、在线服务或者面授课程，每一项 O’Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O’Reilly Radar 博客有口皆碑。”

——*Wired*

“O’Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——*Business 2.0*

“O’Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O’Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野，并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去，Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

目录

Johannes Schindelin序

在 Git 面世之前，Linux 内核开发多年来一直使用专有版本控制系统 BitKeeper，并取得了巨大的成功。但存在一个问题：一些 Linux 开发者反对使用专有性质的版本控制系统，随即开始了漫长的口水战。正是因为这场冲突，授权给 Linux 开发者免费使用的 BitKeeper 许可证被吊销，于是 Git 应运而生。Linus Torvalds 花了两周时间，放下了 Linux 的工作，想要寻找一个替代 BitKeeper 的方案。由于没能找到一个满意的替代工具，他干脆自己写了第一个非常原始的版本，也就是我们现在所说的 Git：一个用 Unix 风格编写的 shell 脚本拼接起来的小程序。具有讽刺意味的是，Git 的分布式是使用 rsync 实现的，而这个工具的作者恰好就是那个将 BitKeeper 推下历史舞台的 Linux 开发者。

就我个人而言，我最初痴迷于 Git 简洁的数据结构，随后参与了 Git 的移植工作，后来又做了越来越多的改进，包括“交互式变基”（抱歉，这个名字有些晦涩）的发明，以及最后对 Git 的 Windows 移植版本的维护。10 年来，不论是作为跨学科项目的专职程序员，还是高度分布式开源项目的负责人，从事生命科学研究的我几乎每天都要用到 Git。

在巴黎举行的庆祝 Git 十周年的 Git Merge 大会上，我第一次见到了 Emma，她分享了一场名为“教人学懂 Git”（https://youtu.be/xYhHi8yK-Is）的激情演讲。这个演讲给我留下了很深的印象，它展现了 Emma 广泛的技能以及在教学和项目管理中的丰富经验。

这本书视角独特，它强调了 Git 如何能够促进团队协作，让我收获颇丰。这本书讲的是那么简单明了，而多年来，我一直沉溺于技术细节，教授 Git 时总是从头开始面面俱到，这可能是最令人受挫的教学方式之一了。这本书重点介绍工作流和角色之间的沟通，引导读者理解实际项目中遇到的真实需求。了解这些知识后，你会学到有趣的部分：如何使用 Git 来支持你的需求。

正如 Emma 的演讲一样，她的写作风格也令人愉悦，使这本书兼具教育性和趣味性。这本书给我的日常工作带来了有价值的见解。不论你在日常工作中担任什么角色，本书都不仅仅是一本手册。无论是探索团队协作的不同方式，还是探索现代版本控制系统帮助推进项目的方法，就让这本书来激发你释放 Git 的全部潜能，为你的工作提供支持吧。

Johannes Schindelin 博士
Windows 端 Git 维护者
2015 年 8 月于德国科隆市

Mark Atwood序

版本控制的重要性怎么强调都不为过。

我认为它的重要性不亚于黑板和书本的发明，因为它将众人的力量聚集在一起，从而创造了更大的价值。

在我的职业生涯中，我看到在软件开发中，版本控制系统从最初遭到抵触到现在遍地开花，也看到基础技术带来的大跃进，其中每一次飞跃都提升了我们创造的工作价值，加快了创造的速度。我们正在以更快的速度，和更多的人一起，完成更多的工作。

Git 带来的最近一次飞跃，对我们的工作流几乎没有作出任何限制。因此，我们需要探索和分享适合自己及团队的工作流，而不是使用以往为机器设计的难用的工作流。这本书介绍了其中一些工作流。我相信你在未来会遇到更多的工作流。

教育的重要性和难度也不言而喻。所谓的教育不只是简单地死记硬背或反复训练，而是更深入的教化：如何以特定方式思考，理解为何要如此思考，以及如何告诉别人你的想法。

正确地使用版本控制系统就是一种思考方式：用精益求精的软件开发所要求的深度和严谨来建模、记忆、交流。如果没有那样的理解，Git 不过是使用一些死记硬背、充满未知危险的“神奇咒语”。有了那样的理解，Git 会变得几乎无法察觉，留给你复杂的命令背后的模式，也就是软件的魔力根源。

这本书会帮助你提升自己对 Git 的理解，并学以致用。

Mark Atwood
惠普公司开源主管
2015 年 8 月于华盛顿州西雅图市

献给 Joe Shindelar。谢谢你！

前言

在将近二十年里，我在个人和团队开发中或多或少都采用了分布式的工作方式。我的第一份有报酬的 Web 开发工作是在 20 世纪 90 年代中期。那时，我维护文件版本的方式只是通过修改文件名来标记一个新的版本。我的工作区堆满了这些文件，它们的扩展名很不寻常，像 v4.old-er.bak 这样的名称到处都是。记录我的工作可不是件容易的事情。在一个对我来说很有挑战的项目中，我不得不用上了之前修改论文时的方法：把要修改的 Perl 脚本打印出来，将这些纸装到活页夹里，然后在脚本上用不同颜色的笔做记号，最后把改动转录回文本编辑器。（要是有照片可以分享就好了。）我通过翻阅活页夹来找到之前的脚本，从而记录版本。我完全不知道该如何搭建一个真正的版本控制系统（version control system，VCS），而仅仅执着于让正确的内容不会因为重构失败而丢失。

当开始和其他开发者一起工作时，不管是做开源项目还是客户的项目，我都不是第一个加入的开发者。在我加入时总是已经有了某个版本控制系统，一般来说是并发版本系统（concurrent versions system，CVS）。它不是最易用的系统，但和我写满了改动的活页夹比起来，它的可扩展性对于分布式工作团队来说显然强多了。很快我就开始对提交信息重视起来，并珍视这种能够审阅其他同事的工作的便利性。它促使我观察其他人提交到仓库中的代码。我可不想让他们认为我在偷懒！

与此同时，我在几个不同的社区大学里教授 Web 开发。2004 年在汉博学院由 Bernie Monette 设计的一个为期一年的项目中，我第一次有机会教授版本控制。整个班级被分成了几组。在第一个学期中，学生草拟了一个网站开发计划。在第二个学期中，团队被打乱了，新的团队被要求构建上一个团队所描述的网站。在第三个也是最后一个学期中，分组被再次打乱，最终的任务是对建好的网站进行 bug 修复和质量保证。每个团队都被强制要求使用版本管理来记录他们的工作。这些先前没有编程经验的学生并没有对使用版本控制感到兴奋，反而觉得这妨碍了他们的工作。但事情也变得更简单了，因为他们从来都没有意外地覆盖其他同学的工作。这件事在很大程度上教育了我，使我知道如果一个工具看上去并非必需，应该如何激励人们去使用它。

在那门课后的十年里，我学到了很多版本控制的教学方法，以及在成人教育中获得的最佳实践。这本书正是我学到的精华，讲述通过版本控制进行有效协作的方法。在整本书中，我都鼓励你因地制宜。不会有“Git 警察”突然出现，告诉你“你做错了”。也就是说，我

会尽我所能地告诉你“Git 风格”的工作方式，当你想要开始在团队中实践，或是获得自我提升时，给你一些指导。使用“通行”的工作方式有助于你和其他之前使用过类似技术的人找到共同语言。

这本书不是写给所有人的，而是写给那些热爱提前计划，然后遵循明确线路的人。我希望本书至少有助于弥补当前 Git 资料中的空缺。与其说这是一本软件指南，不如说它是一本团队协作指南。如果你的团队觉得书中内容有困惑之处，我希望你能够发送电子邮件至 emma@gitforteams.com 来告诉我；如果你觉得它有用的话，我希望你能让全世界都知道。

致谢

几年前，在布拉格一个墓地边上的小酒馆里，我向 Carl Wiedemann 请教了很多关于 Git 的问题。谢谢你，Carl。你的热情激励我化挫折为动力，避免别人重复我在学习 Git 时所走过的弯路。

和 Joe Shindelar 一起工作的时光是我宝贵的财富，那是我在十年的自由职业之后的第一份工作。Joe，你对卓越的追求提高了我在工作上的自我要求。我很感激你的耐心和领导。这本书起源于我们关于领导力和团队结构的谈话，以及我们为 Drupalize.Me 团队创建的 Git 文档。谢谢你。

O’Reilly 找到了杰出的 Christophe Portneuve 作为我的技术审稿人之一。Christophe，谢谢你在我编写前几章时的耐心。你的反馈是非常宝贵的。我很感谢我们在 Git Merge 大会上的讨论，这场对话帮助我厘清了书中用到的概念，这让我有一个崇高的目标，那就是转变人们学习 Git 的方式。我希望你参与的这本书会让你引以为荣。

Bernie Monette、Martin Poole 和 Drew McLelland：你们为我提供了一个平台，让我通过你们的项目来完善我对版本控制的理解。

Lorna Jane Mitchell，感谢你不倦的鼓励。谢谢你和我分享你自己的 Git 工作。这激励着我要对自己提出更高的要求。

推动我完成本书的“燃料”来自 200 Degrees Coffee，一家诺丁汉郡的烘焙坊。我首选的饮料是这家烘焙坊或者司法博物馆画廊前的 Divine Coffee 制作的馥芮白。谢谢 200 Degrees Coffee 和 Divine Coffee 为我提供了休憩的去处，让我想待多久就可以待多久。

致 O’Reilly 大家庭：你们对我所有的请求（以及错过的截稿期限）都处理得非常棒。谢谢 Rachel、Heather、Robert、Colleen、Brian、Josh、Rebecca、Kim，还有数不清的幕后英雄，是你们让这本书的出版成为了现实。

致 Git 核心社区：谢谢你们邀请我参加 2015 年的 Git Merge 大会。你们接纳了我在台上关于探索 Git 教学新方法的激烈演说。你们真心听取了我的建议，然后改进了 Git 的使用体验。我期待在这个优秀的社区里参加更多活动，这里有你们一直以来默默的奉献。

我也要感谢我的审稿人：Diane Tani、Novella Chiechi、Amy Brown、Blake Winton、Stuart Langridge、Stewart Russell、Dave Hammond、John Wynstra、Chris Tankersley、Mike Anello、Piotr Sipika、Nancy Deschenes、Robert Day、Dave Hammond、Sébastien Simard、

Tobias Hiep、Nick Gard、Christopher Maneu、Johannes Schindelin、Edward Thomson、mattj. sorenson、Douwe Maan、Sytse Sijbrandij、Rob Allen、Steven Pears、Laura Lemay。你们的反馈非常宝贵。

致我的伴侣 James Westby：感谢你耐心地等待我完成“最后一件事”。若是少了你的支持和鼓励，这本书也就不会面世了。

引言

本书采用以人为本的方式讲解版本控制。我不想从 Git 的历史讲起，而是在一开始先概览各式各样的团队协作方法。接下来，我们会绕回到 Git 命令，确保你在敲下命令键时总是知道为什么。通过使用特定的工作流，有时你可以节省自己未来的时间（并减少困惑）。这些说明会给你一个宏观的理解，告诉你当下的工作是如何影响到未来的工作的，也希望你能清楚为什么有些人如此执着于他们的 Git 方法论。

第一部分主要面向管理者、技术团队的负责人、首席技术官、项目经理，以及需要制定团队工作流的技术型项目经理。

优秀的技术源自优秀的团队。在第 1 章中，你会了解为什么要创建一个优秀团队。学完这一章，你将能分清团队中的所有角色，组织富有成效的会议，通过关键词识别出和团队脱节的成员，并使用策略来培养团队成员间的认同和信任。

尽早为项目设定预期。在第 2 章中，你会学到用于允许和拒绝访问 Git 仓库的不同权限策略。是否应该允许团队成员跳过评审直接将工作存至仓库？这是不是更像一个信任和被信任的问题？这两种方法各有其优点，这一章将详细介绍它们。

带着清晰的目标工作。在 Git 中，你将会使用分支来分离不同的工作。第 3 章向你展示如何使用分支来隔离团队中运用的不同构想。当然，你还需要知道如何将分散的工作拼成一个完整的软件。这一章介绍了一些常见的分支策略，其中包括 GitFlow。

记录有助于日后工作的文档。第 4 章是第一部分中所有概念的汇总。你会学到如何创建自己的文档，并浏览一个简单的软件产品的创建和部署的全过程。

第二部分主要面向开发者。在这一部分中，你将会学到 Git 命令究竟是如何工作的（终于讲到这儿了）。如果你很着急，希望立刻开始编写代码，只需要从第二部分开始看起，看完之后再回到第一部分。

用实战技能武装自己。第 5 章介绍分布式版本控制的所有基础概念。在这一章中你会学到如何创建仓库，以及通过提交、分支和标签在本地记录你对文件的更改。

学会从错误中恢复。第 6 章讲解如何浏览历史版本，包括如何修复提交、从时间线上移除

提交以及变基。

和团队成员协同工作。现在你已经对浏览自己的仓库的历史游刃有余，是时候和别人一起协作了。第 7 章将告诉你如何跟踪远端更改，将代码上传至一个共享仓库，并将其他成员的更新同步到你的本地仓库。

通过同行评审，分享出色完成工作时的荣耀和责任。在第 8 章中，你将会学习在团队中实践代码评审的流程。我们还会讲到在常见的评审方法中使用的命令，以及为自己的团队定制时的一些建议。

探索项目历史，寻求问题解决之道。在第 9 章中，你将学会用一些高级的 Git 方法来跟踪 bug。不过，不要害怕！这些命令不会比之前的命令更难学。

第三部分是最后一部分，介绍一些市面上流行的代码托管系统。这部分内容既适合管理者也适合开发者。

通过开放的协作促进社区的成长。第 10 章讲解在 GitHub 上建立和维护一个开源项目的方法。

想要编写优良的代码，团队必须拥有自己的仓库。在第 11 章中，你将学习如何在私有仓库中协作。这一章尤其适用于那些希望建立私有仓库，但没有充足资金购买 GitHub 上私有仓库的团队。

良好的约束可以营造更好的氛围。在第 12 章中，你将学习如何托管你自己的 GitLab 实例，并在上面运行项目。这对于防火墙内无法接触到公共互联网的开发者来说尤为有用。

本书不是面向所有人的。对于喜欢自己折腾和探索的读者来说，阅读本书未免会感到沮丧。反之，本书适合有些畏惧未知事物的读者。

补充资料和更清晰的流程图可以在本书的官方网站（http://gitforteams.com/）上找到。

排版约定

本书使用了下列排版约定。

- *楷体*
  表示新术语或重点强调的内容。
- 等宽字体（`constant width`）
  表示程序片段，以及正文中出现的变量、函数名、数据库、数据类型、环境变量、语句和关键字等。
- 加粗等宽字体（**`constant width bold`**）
  表示应该由用户输入的命令或其他文本。
- 等宽斜体（*`constant width italic`*）
  表示应该由用户输入的值或根据上下文确定的值替换的文本。

该图标表示提示或建议。

该图标表示一般注记。

该图标表示警告或警示。

代码用例

本书的补充资料（示例代码、练习等）可以从网站 http://gitforteams.com 下载。

本书是为了帮你做好工作。一般来说，你可以在程序和文档中使用本书的代码。除非你使用了很大一部分代码，否则无需联系我们获得许可。例如，使用本书的几段代码写一个程序是不需要许可的。销售或分发 O'Reilly 书中示例的光盘（CD-ROM）是需要许可的。通过引用本书和示例代码来回答问题是不需要许可的。把本书中大量的示例代码并入到你的产品文档中是需要许可的。

我们赞赏但不强制要求注明信息来源。信息来源通常包括书名、作者和国际标准书号（ISBN）。例如："*Git for Teams* by Emma Jane Hogbin Westby (O'Reilly). Copyright 2015 Emma Jane Hogbin Westby, 978-1-491-91118-1."

如果你觉得对示例代码的使用超出了正当引用或这里给出的许可范围，请随时发送电子邮件到 permissions@oreilly.com，与联系我们。

Safari® Books Online

Safari Books Online（http://safaribooksonline.com/）是应运而生的数字图书馆。它同时以图书和视频的形式出版世界顶级技术和商务作家的专业作品（https://www.safaribooksonline.com/explore/）。

技术专家、软件开发人员、Web 设计师、商务人士和创意专家等，在开展调研、解决问题、学习和认证培训时，都将 Safari Books Online 视作获取资料的首选渠道。

Safari Books Online 为企业（https://www.safaribooksonline.com/enterprise/）、政府机构（https://www.safaribooksonline.com/government/）、教育机构（https://www.safaribooksonline.com/academic-public-library/）和个人读者提供了一系列的产品组合和价格体系（https://www.safaribooksonline.com/pricing/）。

会员可在一个支持完全搜索的数据库中访问数以千计的图书、培训视频和尚未发行的书稿。发行这些内容的是 O'Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 以及其他数百家发行商（https://www.safaribooksonline.com/our-library/）。要想了解 Safari Books Online 的更多信息，请访问我们的网站（http://www.safaribooksonline.com/）。

联系我们

请把对本书的评价和问题发给出版社。

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

O'Reilly 的每一本书都有专属网页，你可以在那儿找到本书的相关信息，包括勘误表[1]、示例代码以及其他信息。本书的网站地址是：http://shop.oreilly.com/product/0636920034520.do

对于本书的评论和技术性问题，请发送电子邮件到：bookquestions@oreilly.com

要了解更多 O'Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：

http://www.oreilly.com

我们在 Facebook 的地址如下：http://facebook.com/oreilly

请关注我们的 Twitter 动态：http://twitter.com/oreillymedia

我们的 YouTube 视频地址如下：http://www.youtube.com/oreillymedia

电子书

扫描如下二维码，即可购买本书电子版。

注 1：本书中文版的勘误，可到 http://www.ituring.com.cn/book/1779 查看和提交。——编者注

第一部分

制定工作流

刚开始教授编程时，我们常常会编写一个简单的小程序，来演示一组特定命令的输出。成人学习者常常会想“这有什么用呢”，他们不知道如何将这些命令用于实际场景。本书从宏观视角展开，讲述以不同方式组织工作流将会如何影响团队协作方式。如果你倾向于用命令思考，请跳过这个部分，从第二部分读起。然后，当你开始问自己“这有什么用”的时候，再回到这部分，就能明白基于 Git 的日常工作会如何影响未来的协作。

这一部分对于那些监督工作是如何完成的读者来说最有用。这些人大多担任管理角色，其中可能包括技术团队的负责人、首席技术官、经理、项目经理以及技术型项目经理。

第 1 章

团队作战

我教授版本控制已经超过十年了。参加我的现场研讨会的大部分人都疲于处理办公室政治问题，而不是技术问题。当然，这些问题不尽相同。或许他们苦于让同事明白版本控制有多么重要，或许他们想要明确责任关系，又或许他们被团队选出来去搞清楚团队工作流中的乱象。不管是什么问题，先理解并解决背后的人际关系问题可以使 Git 的学习和使用更加容易。

完成本章学习之后，你将具备以下技能。

- 识别一个完整团队中的各个角色
- 组织一个富有成效的会议
- 通过关键词识别与你的团队脱节的成员
- 使用策略来培养团队成员彼此间的认同和信任

在开始前，你必须理解你的团队以及软件需求。如果一开始团队就充满了信任和彼此关爱，当你计划用 Git 命令来达成目标时，将会发现自己一身轻松。在充满信任的团队里，当有人遇到困难时，你们可以互相帮助，人们在需要帮助时也会更加坦诚。当人们感受到支持，并理解为什么要用这些 Git 命令时，他们会更可能让 Git 为他们所用，而不是死记硬背几个命令，祈祷自己用对了。

1.1　团队成员

在小团队中，可能一个人就承担了很多角色。紧跟小团队中每个人的日常工作相对容易。然而，在大团队中，角色可能分散在不同部门。那些对代码库进行用户验收测试的团队，可能从来没有和被测试产品的设计师和开发者说过话。两种团队都可能遇到问题：如果没有足够的背景知识，却被提出了更高的要求，最后注定会有所遗漏；团队之间人为的屏障

总是会增加他们之间的紧张关系。在开发代码时，这样的隔阂可不是什么好事。

你是否听说过“以终为始”这句话？当我构建软件时，总是在替别人构建。即使我拼命回忆，也想不起来自己曾经出于自娱自乐的目的开发过某个产品。我不是天生的黑客。我被软件吸引，是因为它能带给别人价值。每次我坐下来解决一个问题时，想的都是给用户提供更好的体验。我希望避免反复，也希望保护用户的安全。我希望他们感觉到自己心灵手巧，而非笨拙。如果在我和用户之间还隔了客户，我有时还需要改变客户思考问题的方式，以便满足他们的商业目标，同时保证终端用户的良好体验。每当我们坐下来工作时，应该从描述希望为用户解决的问题开始，也就是从*用户故事*（user story）开始。

接下来，在测试驱动开发流程中，你将会编写验收测试，界定如何知道问题已被解决。声明会依据编写用途被自动化测试套件、质量保证（QA）团队或是同行评审员使用。如果提前与测试团队商定验收测试，那么开发者会更清楚工作的产出应该是什么样的。一般来说，测试应该描述需要解决的问题，而不是规定将要使用的技术。

测试流程应该包含安全性评审。规模更大的公司有幸拥有专门的安全专家。让这些专家尽早介入这个流程，请他们教你如何编写安全的代码。如果你的 QA、安全和开发团队是分散的，在一开始将他们聚在一起，这会使测试流程变得更加有趣，因为开发者力图提供完美的代码，而测试团队力图挑刺。

如果部署不由你负责，同样让运维团队尽早介入。保证你的开发环境和最终的产品环境越接近越好。理想情况下，你应该使用构建脚本（build script）来尽可能自动复制环境。你可以选择使用 Docker（http://www.docker.com/）和 Vagrant（http://vagrantup.com/）来创建环境副本。和运维团队一起，使用诸如 Chef（https://www.chef.io/chef/）、Puppet（https://puppetlabs.com/）或 Ansible（http://www.ansible.com/home）这样的工具创建管理配置文件的基础设施。

讲到开发的技术栈，如果你在使用开源软件，请了解一下你将要使用的产品的开发社区。我们很少遇到新的问题，而有的人可能已经遇到过你的问题。在代码社区中寻找导师，同时指导别人。打破团队的边界，走出办公室，可怕的问题会变得简单得多。

当促进大公司中部门间的协作时，可以减少代码在原地闲置的时间。闲置的代码会以各种方式耗费你的金钱：新特性的代码可能阻碍你赚更多的钱；修复 bug 的代码则可能阻碍你停止损失。闲置的代码慢慢就不新鲜了。代码等待评审的时间越长，它可能偏离你的主分支越远。它偏离得越远，同步并预发这些工作就越麻烦。

最后，我们会审视自己的团队。技术架构师负责规划解决方案应该如何实施。架构的决策应该有文档记录并尽可能共享。架构师可能也是编码团队的一员。编码团队可能由前端开发者、后端开发者、设计师和项目经理组成。我有时候也和商业分析师一起工作。如果你在敏捷开发环境中工作，可能还需要一个敏捷专家和一个产品负责人。

我倾向于在这样的环境中工作：无论哪里有需要，每个人都愿意伸出援手。自我管理的团队通常彼此非常信任和尊重。不过，这种状态是需要你努力构建的。共识驱动的开发最适合小规模的内部项目，但这并不意味着你不能在其他地方尝试协作。管理项目时，我喜欢让开发者选择他们想做的工单。这增加了他们的自主性，如果需要，让开发者从任务中解脱片刻。不过，我也发现有些人喜欢别人替他们分配好工单。

没有一种方式可以组织所有团队或管理所有项目。一个充满斗志和凝聚力的团队，其秘诀是尊重团队中的每个人，只要有可能，就根据他们的喜好来改善流程。

1.2 思维策略

团队中的每个人都有自己的工作习惯，不同的工作方式适用于不同的场景。没有一种所谓正确的做事方法。如果你能共享每个人高效产出的策略，拥抱差异，将会使你的团队更加强大。我知道我总是在寻找更高效地工作的小窍门，我渴望能了解让人们愿意全身心投入一件事的原因。

几年前，我曾接触过一个领导力培训项目——Bob Wiele 的"领导力与取得成功的四个维度"（http://onesmartworld.com/），其中介绍了一系列的思维策略。这个项目帮助我明白为什么一些类型的活动会让我觉得很享受，而另一些却让我感到无聊。它还教给我很多，例如怎样组织会议，怎样与别人通过交流来获得工作需要的资源。如果团队中每个人都能听懂其中的术语，那么这个系统将最有成效，不必说服其他人参与，你也可以利用这个系统。它将思维拆分成了三个维度：创造性思维、理解性思维以及决策性思维。第四个维度是个人精神，用于标示一个人可能的参与程度。我认为它像是一个音量旋钮或调节器，用来控制这些角色扮演游戏中的成员。

每个人对思维策略的不同偏好很快就会使团队出现分歧。如果我正在进行头脑风暴来解决 Git 中的合并冲突，而你告诉我本不该使用变基（rebase），这时我们就产生了分歧。我正在用我习惯的思维方式来解决问题，而你用你惯用的思维方式打断了这场对话。注意，这些偏好会帮助我们在完成以下事项时加强协作：构建新功能，组织更高效的代码评审，以及打造更健康快乐的团队。

在借助这三个思维维度的会议中，我们最容易引入配合偏好与搁置偏好的概念。确保会议的成果可以帮助人们明白在会议中采用哪种思维策略好，这种思维可以接着被带入到代码评审中，并帮助遇到 Git 流程问题或是在共同开发的产品上遇到具体实现问题的同事。

让我们更具体地审视一下上面提到的几种思维策略。

*创造性思维者*最大的财富是能够找到无法预见的问题解决方案。如果任其发展，那么创造性思维者有时会花费太多时间思考不同的解决方案，而不是专注于一个想法并付诸实践。创造性思维者具有以下特征。

- *预见未来*
  预见另一种未来（可能好也可能坏）。有助于需要长期战略的工作。

- *另辟蹊径*
  略微偏离现状，或换个角度审视现状。

- *头脑风暴*
  有助于突破问题。头脑风暴基本等同于天马行空地思考问题的能力。它要求你脱口而出，而不用担心说错。

- *灵光一闪*

  头脑风暴需要“费力”思考，而灵感来临时你并没有在思考这个问题，或许你正出门散步或正在洗澡。

- *勇于质疑*

  质疑现状。叛逆者、童话《皇帝的新装》里指出皇帝光着身子的那个小孩，都是勇于质疑的例子。

- *保持专注*

  排除干扰，专注于某项任务。在不被打断的工作流中，你可以更深入地思考问题，并更全面地理解问题。

以下是创造性思维者的一些惯用语。

- “我们能不能试着……”
- “我知道我们已经做完了，但……怎么办？”
- “我的天！我突然有了一个绝妙的想法……”
- “你有没有想过这样做？”

通过培养团队的创造性思维，你会拥有掌握问题的全新方法，进而得以改善工作流并解决更大的问题。

下一个类别的思维是*理解性思维*。它可以分为两类：理解信息（分析型）和理解他人（同理心）。分析型思维者最大的财富是能够发现规律、阐明现状。科技行业倾向于吸引具备这些思维策略的人。分析型思维者具有以下特征。

- *评估现状*

  调查所处的环境，搜集尽可能多的信息。

- *阐明现状*

  通过搜集信息和提出问题来弄清当前的情形。

- *善于组织*

  合理、系统地组织数据、人员、资源和过程。

- *敏锐感知*

  感知和理解当前情景下的情绪。

- *产生共鸣*

  包容和理解他人的想法、情绪和处境。

- *善于表达*

  选择合适的情绪和言语向听众传递正确的信息。

以下是分析型思维者的一些惯用语。

- “你的意思是……吗？”
- “解释一下……”
- “你能告诉我怎样去……？”

- “这和……有关吗？”
- “我做了一张电子表格……”
- “那一定很可怕吧！”

最后介绍“敢作敢当”的思维策略：决策性思维。有些人不喜欢反复斟酌。他们想要一个迅速的决策，然后进入下一步行动！决策技能帮助团队找到问题的根源，然后决定如何继续。决策性思维者的弱点是缺乏耐心。他们通常在创造性思维者提出可能的最佳方案或者完成缜密的分析之前，就跳到了其他人的前面。决策性思维者有时会被误解为表现消极。这是一种误解。利用他们的能力快速找到最佳方案，这是弥足珍贵的。决策性思维者具有以下特征。

- 分清主次
  看透本质，也就是问题中最重要的部分。
- 善于总结
  寻找符合逻辑的决策或方案，以最好的方式继续前行。
- 验证结论
  提出问题，排除劣质的方案和无用的信息，审慎评估并确保决策是最佳的。
- 身体力行
  依靠亲身体验引导决策的制定和问题的解决。
- 价值驱动
  依靠自己的核心理念分辨事物的好坏与对错。
- 相信直觉
  不依靠信息，而是用直觉和本能引导决策的产生。

以下是决策性思维者的一些惯用语。

- “我已经准备好进入下一阶段的……”
- “不行。我们已经决定了……”
- “我不知道我为什么会想到这，但是……”
- “上次我们尝试这么做的时候……”
- “我认为问题的本质是……”
- “直觉告诉我……”

1.3 团队会议

我的整个职业生涯几乎都是在分布式团队中度过的，我和同事不在同一个办公室。我们在同一个时区都是少有的事。这些经历让我形成了不少良好的沟通习惯，我常常将这些习惯当作是理所当然的。如果你在工作中使用的是约定好的方法，那么你的团队或许已经有一套推进项目的会议模式。

你的项目以及其中的每个子模块，都应该有开头、主体和结尾三个部分。Dave Gray、Sunni Brown 和 James Macanufo 合著的 *Gamestorming*（http://shop.oreilly.com/product/97805

96804183.do）一书详细阐述了“开头—主体—结尾”这一流程。这个流程还被教师应用于课堂教学：教师首先告诉你学什么，让你参与学习，然后总结你学到的知识。

回到会议安排上来，你应该在脑中熟记这个规律：开场、参与、总结。这个规律最适用于会议。我经常看到一些会议，准备了讨论话题的大纲，但最后的结果却差强人意。例如，项目刚开始时，团队正在参与*构思会议*，创造性思维者的参与最为积极且成效显著，如下所示。

时间表：构思阶段总时间为 45 分钟

- 辨别问题本质（10 分钟）
- 头脑风暴，寻找解决方案（25 分钟）
- 整理想法（5 分钟）
- 挑选至多三个想法进行验证（5 分钟）

提前制定会议目标非常容易，这样就可以用一些自由时间来讨论问题。

1.3.1 项目启动

项目启动会是一个混乱的时期，尤其当你召集的是一个新团队，而团队的成员在工作上没有交集时。如果有可能，请召集一个全员参与的启动会议。对分布式团队来说，时间和金钱的代价会是异常昂贵的。

面对面的会议更佳

理想情况下，启动会议是面对面进行的。如果难以实现，试着将人们聚集到尽量少的几个地点，然后使用视频电话召开会议。

当所有人共处一地时，你可以充分利用他们共度的时光。你可以借助白板、活页挂图和便利贴，用动作来表达你的想法。看到大家共同作出的决策是非常令人高兴的，它有助于激励团队一起参与到项目中来。

1.3.2 追踪进展

项目开始后，你会希望定期与团队开个会。当你在分布式团队中工作时，逃避问题是非常容易的事。跟不上进度是一件很令人难堪的事，而且通常是一个复杂的问题。保持沟通是一个处理此类问题的好习惯，但这并不意味着要将所有时间浪费在开会上。成功的团队总是有着明确的目标。我喜欢一周一次的、非常短小的冲刺周期。在这么短的时间内很难隐藏什么问题。不过，它和微观管理没有关系。它的目的在于保证项目持续推进。以下每个会议都有一个与项目相关的具体目标。

- *冲刺计划会议*

  作为项目经理，我发现有两种类型的员工：其中一种员工随时准备接手新的工作并对做完的工作负责，而另一种员工倾向于别人为他们安排好工作。那些希望别人为他们安排工作的成员经常寻求帮助，来弄清楚他们能胜任哪些任务，以及从项目整体来说哪些任务有最高的商业价值。冲刺计划会议可以邀请全员参与，而如果你不希望在冲刺计划会

议上耗费过多时间，也可以仅让跟客户打交道的成员和高级开发者参与。

- *承诺会议*
  这类会议应该挑选每周几天中的同一时间召开。会议的成果是团队成员针对他们工作中作出的“承诺”进行汇报。他们不应该只汇报“今天在做什么”，还应该汇报“下次开会前计划完成哪些工作”。会议应该是采用“不责备，不让人羞愧”的轮流发言方式，每个人汇报进展的时间不应该超过三分钟。更大的具体问题可以留到后续的会议中去讨论。在 Scrum 的用语中，这类承诺会议称为“站立会议”，参会者一般站着参加会议。我发现“站立”对于那些没有接受过 Scrum 训练的团队来说并不准确。使用适合你的团队的术语，但要确保你能从会议中获得有价值的信息。

- *深入研讨会议*
  任何承诺会议之后还需要深入讨论的问题都应该安排一个后续的深入研讨会议。在理想情况下，你的团队将使用一个日程系统，比如“谷歌日历”（http://google.com/calendar），成员可以在上面查看同事的日程，并很容易找出一段空闲时间来安排后续的讨论。一般来说，我每周会保留一两个 45 分钟的时间段用于深入研讨会议，紧接在两个 15 分钟的承诺会议后。只有相关的人员需要参加深入研讨会议，虽然我们欢迎任何人加入。

- *冲刺演示会议*
  团队应该每周找一个时间一起展示工作。在演示会议中，每个取得成果的成员应该列出完成的工单号，并展示工作成果。每周安排一次演示会议形成工作“永远即将完成”的文化，在这种文化中，工作被分成易于执行的小块。这类会议提供了一个绝佳的机会，让你发现新想法，分辨可能需要文档记录或后续修复的 bug，或者是讨论下一个冲刺中必要的流程改进。由于团队的凝聚力及沟通水平不同，你或许会觉得这些会议是不必要的。但如果你发现越来越多未完成的功能通过了代码审查，或是优秀的工作没有得到重视，或是发现你的团队没有经常相互求助，那么，是时候引进每周的演示会议了。Google Hangouts（http://www.google.com/+/learnmore/hangouts）和 GoToMeeting（http://gotomeeting.com/）非常适合这种会议。

- *冲刺回顾会议*
  在每个冲刺结束的时候，你应该召集团队一起讨论工作流程。找出运转良好的部分以及需要改进的部分。我有一个行之有效的方法，即让每个成员用下面这些提示语作答：我希望；我想要；我担心。这个会议应该只邀请核心成员参加。会议时间可长可短，但对小团队而言大致需要一个小时。

在一个分布式团队中，你可能还需要安排一些定期的社交电话会议。Lullabot（http://lullabot.com/）是一个完全分布式的公司，拥有大约 50 名员工。它将下面几个与项目无关的电话会议加到了日程中。以下会议的目的是培养成员之间的同理心。

- *全公司的站立会议*
  每周举行一次电话会议，通过抽签选出发言者，每人用不超过两分钟的时间聊一聊他们的工作和业余生活。当公司的规模还小时，每个人都要在这个会议上发言。随着公司规模增长，这个抽签的系统便应运而生，一对一的电话会议也加进了日程。

- 一对一会议
  通过抽签选出两三名员工，找个方便的地方交流各自的生活、兴趣等任何事。

最重要的是，这些电话会议只有语音，成员可以在打电话的时候做别的事情（往洗碗机上装盘子，如果手机信号够好，甚至可以在室外活动）。

1.3.3 培养同理心

当你在分布式团队中工作时，将代码团队中的成员视为“资源”而不是个体会容易得多。培养团队间的信任关系需要有意识的努力。彼此信任而不是畏惧的团队将会诞生更多灵感，也将更愿意在困境中寻找合适、创新的解决方案。

提升团队的同理心的第一步是恰如其分地关心你的同事。你不需要成为每个人的心理治疗师，但花点时间和他们聊聊私下的生活会是非常值得的。如果你被认为很有爱心，大家会更加喜欢你，从而增进你们之间的信任。作为技术型的项目经理，经常有人邀请我倾听他们讨论问题。在他们为我介绍问题背景之后，我对问题粗浅的理解可以迫使关注点回到问题的本质，也就是解决方案的来源。但这种对话在新团队中鲜有发生，因为我必须先取得团队中每个人的信任（在他们不知道正确答案时，我不会妄加评判；我会帮助他们专注于解决问题，而不是在他们讨论时只顾自己打字）。

以下是一些帮助你表现出“恰如其分”的关心的方法。

- 搜集故事
  询问人们的日常生活；关于他们正在解决的有意思的挑战；关于你们共事的项目中他们喜欢（或不喜欢）的地方。这不是一个闲聊的机会！这是一个将谈话对象和他们的生活联系起来的好机会。

- 带有目的地倾听
  当你和人们交谈时，专注地倾听。不要同时做几件事情。完整地倾听他们所说的内容。不要打断他们，除非你感到困惑，希望得到解释。一些人是天生的演讲家，总是能滔滔不绝地讲下去。对于这些人，你或许希望预先计划一个结束的时间点。

- 旧事重提
  如果有人和你谈起过他们的生活，时不时问问他们之前的故事有什么变化。他们的女儿还在长牙吗？感冒怎么样了？今天好一些了吗？

我考虑将这个清单视为“同理心的培养”（http://gitforteams.com/resources/cultivating-empathy.html）。每个人能够做到并且也应该和同事保持一定的联系。

1.3.4 回顾

上述会议是总结经验、取长补短的最佳时机。这些会议还应该整理未来可以重用的模板。一段工作的结束会议应该是不责备、不让人羞愧的，人们可以畅谈做得不好的地方。作为项目经理，我很少为我的决定感到后悔。我依靠团队提供的信息来帮助我作出最好的决策。所以在回顾中，我总是很容易避免一些“本来应该、可以这么做”的诱惑。不过，我确实在留意一些面向未来的模式。换言之，明确我们在会议中提出的哪些问题本可以得到

解决，以获得另一些有用的信息（这些信息或许会让我们在未来遇到同类项目时能够作出更好的决策）。

从版本控制的角度来说，项目的终点也是一个巨大的机遇，你可以挑出你最喜欢的工单，记录它们与众不同的原因。或许它提出了一种组织信息的新方式，而你想要重用这一点。看一看 Git 仓库，寻找一些特别好的提交信息，供以后的项目参考。

1.4 Git中的团队协作

如果你尚未接触过分布式版本控制，会看到一些术语贯穿了本书剩下的部分。这些术语在简单的开发者工作流中最容易理解。

每个开发者都有一个本地的仓库副本，即项目中的更改历史的独立副本。为了共享更改，开发者一般会将一份仓库的副本发布到一个集中式的代码托管系统，如 GitHub。尽管如此，正如你将在本章剩下的部分中了解到的那样，会有很多种分享代码的方法。

在仓库的中央副本中，开发者将会创建一个他们可以更改的仓库副本。在 Git 的用语中，这个过程称为*克隆副本*，尽管这个过程也可以称为*派生*（forking）。

在克隆仓库时，软件开发者可以选择将他们的项目副本设置为私有的或公开的。一个私有仓库默认不希望别人直接查看这个副本，而只通过查看主项目来获得被官方接受的更改。另一方面，任何人都可以直接向开发者的仓库的公开副本提交贡献。对于软件开发来说，这是一个更加开放的策略，但可能会让人对哪个副本才是项目的起点感到困惑。

只有通过项目治理才能决定哪一个仓库是最重要的版本。这是因为每个仓库都可以接受更改，并与外界共享更改。项目之间的关系不是一成不变的。你可以在不同的仓库副本间建立关系网络，或者建立一个线性的关系链。但一般来说，软件产品的官方版本称为当前仓库的*上游*。例如，我的博客是通过 Sculpin（https://getsculpin.com/）创建的。我将这个软件的官方发布版本克隆下来，并直接修改这个仓库来编写博文。如果我希望获得软件最新的更改，将会并入*上游*的更改。

> **派生一份新的奶油塔食谱**
>
> 对长期开源软件开发者来说，*派生*（fork）这个术语出现时，往往是一个分裂的社区充满了对项目的失望，一群开发者决定“派生一份新的项目”，然后朝着不同的方向发展。派生简单来说就是偏离主路，就像一条林荫小路，或是我的曾祖母 Austin 的奶油塔食谱。每个派生的分支指向了不同方向。在这个奶油塔的例子中，就是加或不加小葡萄干。你可以在附录 A 中读到我家的派生食谱。

在一个单独的仓库中，我可以存放项目的不同版本。这些仓库内的更改可以通过分支进行追踪。为了从我当前的分支切换到另一个，我将会*签出*（check out）我想要切换过去的分支。（我心中默念“这个太酷了，快签出看看”。）在切换前，Git 将会强制要求我处理未提交的更改，要么提交，要么舍弃。*提交*过程会将我的更改持久存储到仓库，而*储藏*（stash）将会暂存更改，允许你随后拉取你刚刚储藏的工作并重新应用。

手工艺人的储藏物

编织者、裁缝和其他纤维艺术家通常会*储藏*一些纱线和织物。当一个新的项目开始时，我们可能去“储藏室购物”而不是去商店。那些储藏量巨大的人也许会谈到“达到了 SABLE 的状态”（储藏的纱线一辈子都用不完）。我认为这个类比对 Git 的储藏物来说也适用，和手工艺一样，我推荐定期清理储藏物以防虫蛀。如果你是一位编织者，你会喜欢上 Git for Knitters（https://github.com/gitforknitters/gitforknitters）。

并入和发布更改的过程使用了下面这些命令。我从远端仓库*抓取*（pull）更改，然后自动并入我的仓库。这个过程*拉取*（fetch）新的更改并将它们*合并*（merge）进入本地*分支*（branch）的*跟踪*（tracked）副本。不管什么时候，我在我的仓库的本地分支上工作。如果想与其他开发者共享更改，我会将我的工作提交至仓库，然后将我的分支*推送*（push）到远端仓库。

1.5 小结

我最喜欢做的一件事是与一个面临危机的团队一起工作，帮助他们找到一个有趣且充满创造性的全新工作方式。这个过程并不总是一帆风顺的，因为面临危机的团队或多或少都存在一些不信任。在这个过程中有时会有泪水，但如果团队齐心协力，最后的回报是巨大的。

- 一个彼此信任、具有同理心的团队更可能帮助同事学习工作需要的具体 Git 命令。
- 不同的思维策略偏好会使你的团队偏离方向。确保在正确的时间使用正确的策略来减少团队间的摩擦，同时让工作进展更加迅速，带来更多乐趣。
- 通过保持工作的透明，并在关键时刻引入利益干系人，你可以缩短代码的测试时间并减少发现的 bug 数量，从而获得更快的部署。

在下一章中，你将会开始学习项目仓库的治理。

第 2 章

命令与控制

从定义上看，分布式版本控制系统避免了集中式系统的权限控制方式。Git 没有提供内置的规则来帮助你控制代码的访问权限，它只是一个内容跟踪工具。如果你已经习惯了版本控制系统同时扮演守门人和访问控制管家的角色，那么着实会感受到明显的不同。没有集中式权限控制并不意味着你的项目将立即陷入混乱的状态。

在 2.1 节中，你将学到以下内容。

- 原作者、版权及分发许可。
- 领导力模型，为项目的贡献规范定下基调。
- 行为守则，制定一份指南，明确贡献者的预期行为和可接受的行为。

接下来，在 2.2 节中，你会学到如何设置项目的访问权限。2.2 节介绍了下面三个模型。

- 分散贡献者
- 并列贡献者仓库
- 共享维护

学完本章，你将能够自信地为你的团队建立一个访问模型，既让贡献者满意，也确保你能够遵从监管机构的任何管理要求。

2.1　项目治理

如果要我打赌，我会赌你拿起这本书的目的是为了学习 Git。本节介绍的是法律条文。如果你素来没有耐心，那么也许会想知道为什么我要在这个晦涩的主题上浪费宝贵的时间。把这些资料当作一本入门读物，了解自己的作者权利以及项目管理的责任。越来越多的政府和大型企业开始使用公开发布的代码，并选择开源自己的代码。（如今连微软都有了不少开源库！微软加油！）

开发开源软件

在本章中，我会讲到项目运营的要点。如果软件开发者和管理者正在考虑开源他们的项目，他们应该读一读 Karl Fogel 的 *Producing Open Source Software*（http://producingoss.com/）。这本免费图书涉及了开源项目运营的方方面面，既包括宣传、增长，也涵盖法律事务和政治结构。

在本节中，你将会了解到一段代码的归属。然后，当你用 Git 工作时，会看到 Git 允许你追踪仓库中每一份代码的作者。不仅如此，你甚至可以为添加到仓库的每一份代码加上签名。

2.1.1 版权和贡献者协议

版权是在使用和分发作品时排他且可转让的法律权利。世界各地的版权法律的细节不尽相同，但一般来说作者有权复制和分发自己的作品。在开源软件中，版权所有者同意将他们的作品授权给更多人。流行的自由开源软件（Free Libre Open Source Software，FLOSS）的分发许可将在下一节中介绍。

如果作者是有偿完成工作中的产品，作品的版权通常会归属于付款人或赞助人。在美国，这称为*雇用作品*（work for hire），几乎总是针对雇主－雇员关系，且一般针对合约雇员。如果你不确定你是否拥有作品的版权，请检查一下你的协议。如果没有这样的条款，可以去当地的司法机构查询是否已有先例。在美国，根据最高法院给出的定义（http://copyright.gov/circs/circ09.pdf），合约雇员和自由工作者的作品不受该限制，不属于雇用作品。不过，这项条款非常模糊。在理想情况下，最好完善你的合同，明确规定作品的版权归属。

版权只保护作品的具体实现。你不能为一个想法声明版权。你也许听说过*逆向工程*（reverse engineering），这是一种规避某个作者对作品的正当权益的做法。世界上某些地方的司法机构还有一项*竞业限制*（restraint of trade）条款。这项条款禁止雇员（或合约雇员）在一段时间内参与别处的类似工作。这项条款有效避免了雇员在接受新的工作后侵权或仿照为前雇主开发的作品。法庭必须认可这是一个“合理”的约束，限制雇员从事某个特定行业或某些具体工作内容，而不能被扩大地解释为禁止雇员从事任何工作。

在一些法律中，专利确实涉及了发明背后的想法。软件专利尤其受到争议，因为人们察觉到它们在许多情况下扼杀了创新。专利向来不是自动授权的，而是必须在特定地区提交申请的。

如果你代表雇主参与一个开源项目，版权的分配或许会更复杂一些。如果项目的政策规定只接受个人的贡献，而你的雇用者保留了你的作品的全部版权，这会变得尤为复杂，你的公司可能规定了你可以在业余时间做什么工作。（我知道的一些开源项目和公司确实有这样的限制。）我不是律师，因此无法给你法律意见。你能做的只有事前寻求权限或事后达成共识。但是，我可以向你强调版权的重要性，并鼓励你从长远利益考虑最合适的做法。如果你的贡献因为某些原因不得不从一个开源项目中被移除，对你来说就有点可惜了。虽然彻底地坦诚存在风险，但我认为最终这会是值得的。

为了保障自己未来的权利，一些公司在公共项目中放置了贡献者协议。Canonical（http://www.ubuntu.com/legal/contributors）、Chef（https://docs.chef.io/community_contributions.html）、Puppet（https://cla.puppetlabs.com/）、Google（https://cla.developers.google.com/about/google-individual）和 .NET（https://cla.dotnetfoundation.org/）都有自己的贡献者许可协议。协议因公司而异，但它们的主旨相同："如果你选择提交一个贡献，你同意将你的版权转让给这个项目。"正如文字作品的创作共用（Creative Commons）协议一样，同样有一个贡献协议的 Harmony Agreement 模板（http://harmonyagreements.org/）。贡献者协议存在的最根本原因是它允许项目无需个人贡献者的明确同意就可更改项目的分发许可。在开源软件中，这些贡献者协议经常被认为与开源精神背道而驰。但另一方面，如果公司不拥有软件的版权，未来若要做出与法律相关的决策将会是非常困难的。

2.1.2 分发许可

一旦你决定如何处置你项目的版权，接下来要做的便是创建一个分发许可。它将阐述你希望别人如何使用或不能如何使用你的项目。

GitHub 将它推荐的一些流行开源许可（http://choosealicense.com/）编成了一本出色的入门手册。这份手册包括了以下的许可证。

- MIT 许可证（http://opensource.org/licenses/MIT）允许人们在注明原作者的前提下自由使用代码，并且你不需要为衍生的软件负责。jQuery 和 Rails 均使用 MIT 许可证。
- Apache 许可证（http://www.apache.org/licenses/LICENSE-2.0.html）类似于 MIT 许可证，但明确将原作者的专利授权给用户，并要求用户提供变更说明，描述你的作品在之前的版本上做了哪些修改。Apache、Subversion 和 NuGet 均使用 Apache 许可证。
- GNU General Public License（GPL）分为 V2（http://www.gnu.org/licenses/gpl-2.0.txt）和 V3（http://www.gnu.org/licenses/gpl-3.0.txt）两个版本。它是一个共享友好的版权协议，要求作品或衍生品的分发者将源码以相同协议共享。V3 版本与 V2 类似，但进一步限制将作品用于禁止软件篡改的硬件。Linux、Git 和 Word Press 均使用这种许可证。
- 如果你的作品形式不是代码，创作共用许可证（http://creativecommons.org/）可能更适合你的作品。这个协议允许你选择重新分发时是否能够修改以及能否用于商业目的。

你也完全可以选择不使用分发许可。但是，这意味着你在告诉别人：你不介意别人未经明确授权就使用你的作品。

什么时候不要使用分发许可

在大多数情况下，在公共项目中使用分发许可都是一个好的做法。也就是说，有时我不会为一些公共项目添加分发许可。如果我觉得我的作品会被收录到传统出版商的完整的书中，我一般会选择这么做。一些出版商要求你将版权重新转让给他们，并代表你保护作品。（O'Reilly 将所有版权归于原作者。）如果我的作品接受了被一个开源许可保护的贡献，这可能会影响我以后重新分配版权。

如果你遇到了一个没有明确提供许可的公共项目，并且希望使用它们的作品，那么请先联系项目的管理者，让他们为自己的作品添加一个分发许可。

2.1.3 领导力模型

开源软件使人们能够一起构建更强大、更安全、功能更丰富、可维护性更强的系统，而维护软件的重任由大家一同分担。如果你的项目只有一位成员，建立治理文档或许还没必要，但如果你预计会有其他贡献者加入，你应该考虑你希望项目如何运转。

我参与过的治理模型包括以下这些。

- 仁慈的独裁者（BDFL）
  在这个模型中，代码库中任何地方的任何决定都由项目领导者说了算。BDFL 类型的领导者或许不会活跃于每次代码评审中，但最终保留了驳回或撤销某个决定的权力。整个社区任由他宰割。听起来很可怕，不是么？好吧，如果这位独裁者不仁慈，那么确实如此。这个模型已经成功应用于 Ubuntu 等项目（https://en.wikipedia.org/wiki/Benevolent_dictator_for_life）。
- 共识驱动、主管批准
  Drupal 社区使用的共识模型鼓励社区在自己熟悉的领域中寻找合适的解决方案。当社区对解决方案感到满意时，他们将这个 issue 标记为 RTBC［Reviewed and Tested by the Community，通过社区审核与测试，与 Ready to be Committed（即将提交）的英文缩写相同］。Drupal 有独立的工作组负责内容、版权和安全问题（https://www.drupal.org/governance）。
- 技术评审委员会或项目管理委员会
  Backdrop 是由 Drupal 派生出去的一个项目，它在初期就采用了明确的治理模型（https://backdropcms.org/leadership），该模型是基于一个 Apache 项目的项目管理委员会（PMC）模型（http://www.apache.org/foundation/governance/pmcs.html）。

如果你在设置项目的治理方案时需要更多的指导，我推荐 Lisa Welchman 编写的资料，包括她的一本名为 *Managing Chaos* 的图书（http://rosenfeldmedia.com/books/managing-chaos/）。

2.1.4 行为守则

如果一些成员对他人不够友好，有时一些社区不得不拒绝他们的代码。如果不这么做，那么那些社区就会因为其中不甚友好、难以忍受的行为而臭名昭著。你或许会想起你乐于参与的一些社区，并且希望将这种氛围带到自己的项目中。

社区文化是持续强化的行为守则。尽管你或许会希望每个人对彼此都很友好，但总有一天你还是会希望有一本可以参考的条例手册。一份社区行为守则使你能够明确规定项目的参与者应该遵守哪些规定。一些行为守则已经久经社区考验，你或许会希望借鉴其中一份开始起草你的行为守则。

据我所知，Flickr 是第一个使用行为守则的社区，它还特地确保成员知道社区已经有了规

范。我相信自从我第一次阅读这份文档之后，它已经修改了不少内容，你可以在 Flickr 社区规范（https://www.flickr.com/help/guidelines）中读到最新的版本。

Drupal 社区行为守则（https://www.drupal.org/dcoc）是我最熟悉的一份社区行为守则。它衍生自 Ubuntu 社区行为守则（https://launchpad.net/codeofconduct/1.0.1）的一份早期版本[现在已有了更新的版本（http://www.ubuntu.com/about/about-ubuntu/conduct)]，并且启发了 Humanitarian ID 社区行为守则（http://humanitarian.id/code-of-conduct/），这是一个联合国人道主义事务协调厅项目的社区行为守则。

你可以将你的行为守则（Code of Conduct，CoC）文档放在项目网站上。如果你的项目没有一个专门的网站，你可以将你的行为守则放到 GitHub 上的一个 Wiki 页面上。在项目首页上方的选项卡中可以找到 Wiki 页面的链接[1]。

2.2 访问模型

如果你使用版本控制系统已经很长时间了，那么或许会想起 CVS、Subversion 等使用中央仓库的版本控制系统。图 2-1 展示了在 Subversion 的集中式系统中更改是如何发生和流动的。在这个系统中，当你每次想要保存一份仓库中的工作快照时，你和所有人的快照其实都存放到了同一个地方。当你想要共享工作，或提交代码审查时，如果有人刚刚在同一分支更新他们的工作，你可能就无法完成操作。

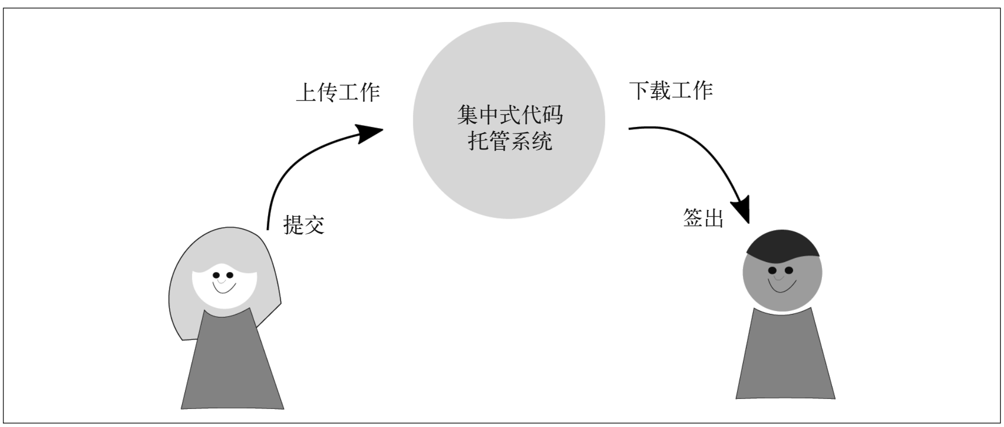


图 2-1：Subversion 中的文件操作

在另一方面，Git 是一个分布式的版本控制系统。也就是说，与强制将更改记录在中央仓库的集中式代码托管系统相比，使用 Git 每个人都能够独立工作，并且在自己本地的仓库副本中提交更改。这意味着其他开发者的修改永远不会强制进入你的作品。相反，你可以决定什么时候引入外部代码以及什么时候共享你的工作。

注 1：由于 GitHub 页面改版，Wiki 的入口从右侧边栏已移至上方的选项卡。——译者注

连接其他仓库

尽管在使用 Git 时，人们经常提到在没有网络的飞机上编写代码这个例子，我认为 Git 真正的优点在于你可以私下里更多地尝试你的想法。你可以创建新的分支，思考代码中的新想法，并且在自己准备好了之后再连接其他仓库。

如果你做过 MBTI 职业性格测试（https://en.wikipedia.org/wiki/Myers%E2%80%93Briggs_Type_Indicator），Git 大概是 INTP（内向 / 直觉 / 理性 / 理解）类型的，而 Subversion 大概是 ESFJ（外倾 / 感觉 / 情感 / 判断）类型的。

每当你开始使用 Git 时，对于你的电脑来说你使用的是集中式系统开发方式，如图 2-2 所示，仓库中的更改全部存放在你自己的机器上。你完成一些工作，然后将这些工作保存到你本地的仓库。当你准备好将工作共享给别人时，连接到远程仓库，将你特定分支的副本推送上去。

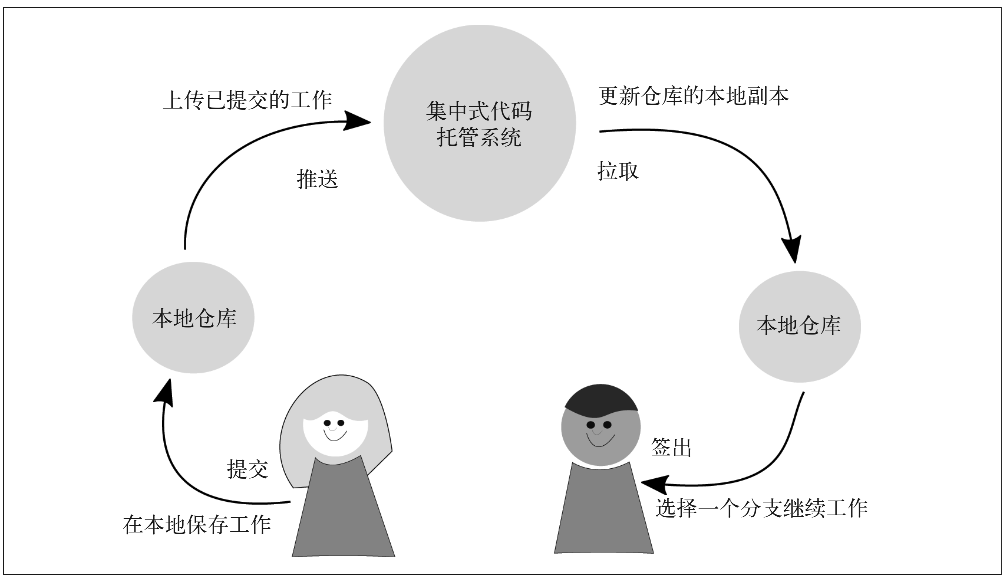


图 2-2：Git 中的文件操作

把工作完全存放在本地就太大材小用了！相反，我们连接到其他系统，通过远程仓库可以共享我们的代码。

Git 不提供访问控制，允许任何开发者拥有仓库的完整读写权限。在最外层，你通过登录控制来限制访问。我在自己的机器上开发，而你没有权限访问，因此你无法更改我的仓库。一旦我将仓库放在共享的区域，比如集中式代码托管服务器，我们需要共同协商如何管理仓库的访问权限。

一些 Git 托管系统，如 Bitbucket，允许细粒度、分支独立的权限控制；但是，大多数托管系统只允许你对每个仓库设置访问控制。也就是说，你要么可以在仓库中任意分支上提交，要么只能通过拉取请求来提交贡献。

在本节中，我们介绍了下面三种最流行的访问模型。

- 单一仓库共同维护模型：团队中的每个人都是维护者，有权限向项目仓库上传更改。
- 并列贡献者仓库模型：提交贡献的开发者创建一个项目的远程副本，等待项目维护者接受他们的更改。
- 分散贡献者仓库模型：代码通过文本格式的补丁包共享。

在本节结束时，你会学会如何将这些方法贯穿起来，建立一个适合你的团队的访问模型。

2.2.1 适合分散贡献者仓库的模型

当 Git 面世时，开源软件项目通常在公开的邮件列表中讨论代码库的更改，而不是在集中式网站中。这个模型如今仍用于 Git 的开发团队。这个模型不太可能适合你的团队的开发，但是，理解这个模型有助于理解使用 `rebase`（第 6 章）和 `bisect`（第 9 章）命令时遇到的一些更高级的概念。

为了在社区中共享工作，开发者需要使用 `diff` 程序创建一个补丁文件。开发者接下来给讨论组发送一份邮件，将图 2-3 所示的补丁文件添加为附件。为了审核提出的更改，邮件列表成员将会下载附件中的补丁文件，并使用系统自带的 `patch` 命令将更改应用到本地代码库。

通过邮件列表共享补丁文件，开发者得以隔离和贡献他们的工作；而通过有效地限制补丁文件共享的内容，审查者可以轻松地看到在共享代码库中两个特定时间点之间的更改。

形式服从于功能

为了让邮件发送补丁文件更加方便，Git 增加了 `am` 命令来支持通过邮件列表发送的补丁。

这个模型如今仍然被 Git 项目自身使用，它仍然使用邮件列表来共享补丁，并讨论哪些功能应该被加入 Git，哪些 bug 应该被消灭。

尽管这个模型看上去有些过时，但它确实具有以下这些优点。

- 你不需要在本地拥有特定的版本控制系统，因为补丁文件不需要在本地安装版本控制软件。
- 开发者可以轻松地使用邮件应用来审查提出的更改。
- 这个模型鼓励*完整*的想法。如果你每次提交更改都需要给一群人发邮件，你更可能会确保所有地方都是正确的，以避免“我又想起一件事”的尴尬。
- 将你提出的更改上传到一个单独的系统，而*不是*上传到代码托管系统，确保让软件项目中的成员来实施审查的过程。换言之，开发者不能直接将更改上传到主仓库，而必须宣布工作完成并等待其他人将它合并。

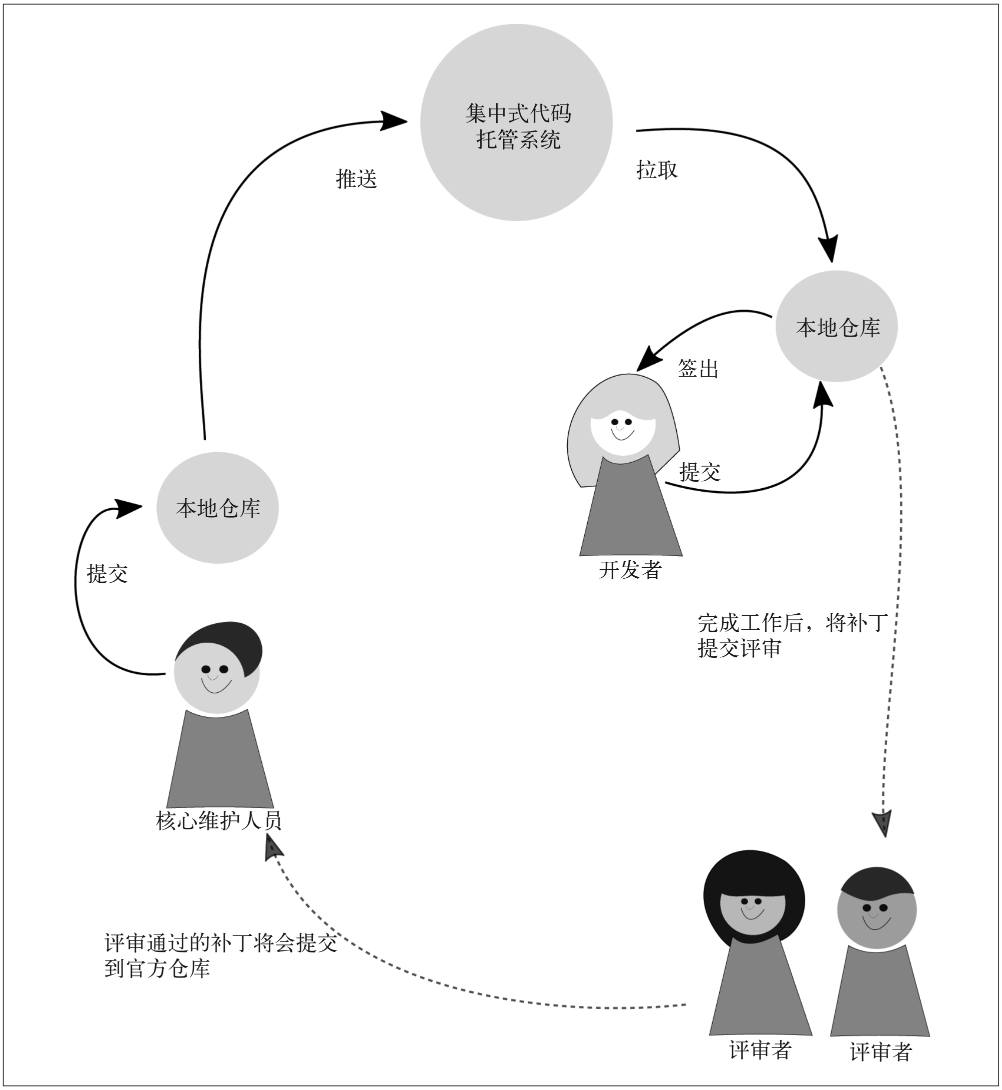


图 2-3：社区对补丁的审查过程

使用分立的仓库不仅仅适用于使用邮件列表沟通的项目。在本书编写时，Drupal 项目正在使用这个模型的一个变种。与使用邮件列表来共享补丁相比，Drupal 项目使用一个自行管理、集中式代码托管和工单 issue 系统。图 2-4 展示了一个包含补丁文件的 issue 屏幕截图。

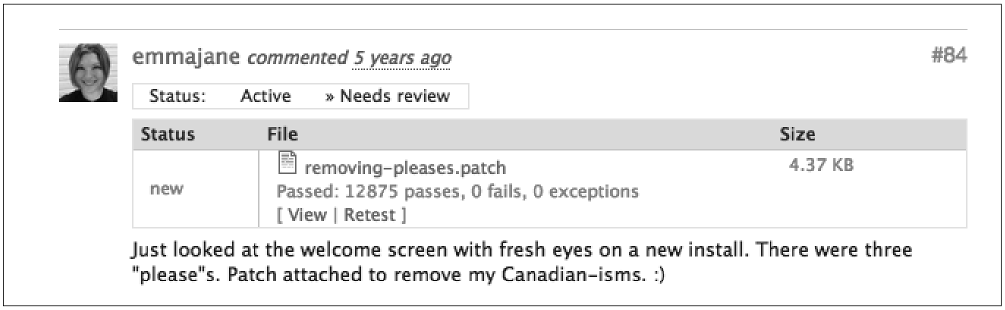


图 2-4：一个带有补丁文件的 Drupal issue 队列

在这个模型中，你可以在共享之前为每个提交署名；但是，如果有多人参与，则很难搞清提交历史中谁做了哪些更改。团队不得不使用补丁格式政策（不论是否签署）以及一个提交消息格式。Drupal 使用了严格的提交消息规范（https://www.drupal.org/node/52287）来确保每个人的工作都被认可。

对于现在开始的大多数项目来说，这个模型是不合适的。但如果你将每个提交视为一个*完整的想法*，这个模型确实有助于理解一些更复杂的命令，比如 `bisect`。这个模型的一个更现代的做法是在单一的代码托管系统中来派生、克隆仓库。

2.2.2　适合并列贡献者仓库的模型

如今，软件开发者不再交换补丁文件，而是通常使用集中式代码托管系统来帮助他们管理打补丁的过程。使用单一的代码托管系统使得在仓库间可编程地创建和提交补丁变得更加容易。补丁的管理方法是每个版本控制系统的独家秘方。Git 的 `pre-commit` 钩子确保在这个过程中能够遵守访问控制。

在并列的系统中，"上游"项目保留了完整的控制，决定谁拥有项目主仓库的写入权限。每个贡献者使用代码托管系统将项目克隆、派生至他们的本地仓库。如图 2-5 所示，贡献者更改本地的副本，然后通过*合并请求*或*拉取请求*提交这些更改。如果你参与的开源项目拥有很多贡献者，你最可能使用的就是这个模型。

GitHub 将这种开发模型推广到了现在很多的开源项目中。我见过一些职能严格分离的内部项目同样应用了这个模型。如果 QA 团队专门负责将最终的代码并入稳定的预发分支，他们可能会在这个模型的基础上做一些修改。如果你使用外部承包商并且不希望他们越过审查直接向仓库提交更改，那么这个模型同样适合你。

Git 和 GitHub 术语的比较

有时候很难知道应该使用哪个术语，因为现在普遍采用的 GitHub 术语并不总是与对应的 Git 命令保持一致。比如，GitHub 的术语 fork 使用 Git 命令 `clone` 创建了一个仓库的副本。因为这本书的重点是 Git 本身，而不是 GitHub 的 Git 实现，所以我们将会使用 Git 命令。我们偶尔会同时用到两种术语，因为有时我们更熟悉 GitHub 的术语，而不是单独的命令。

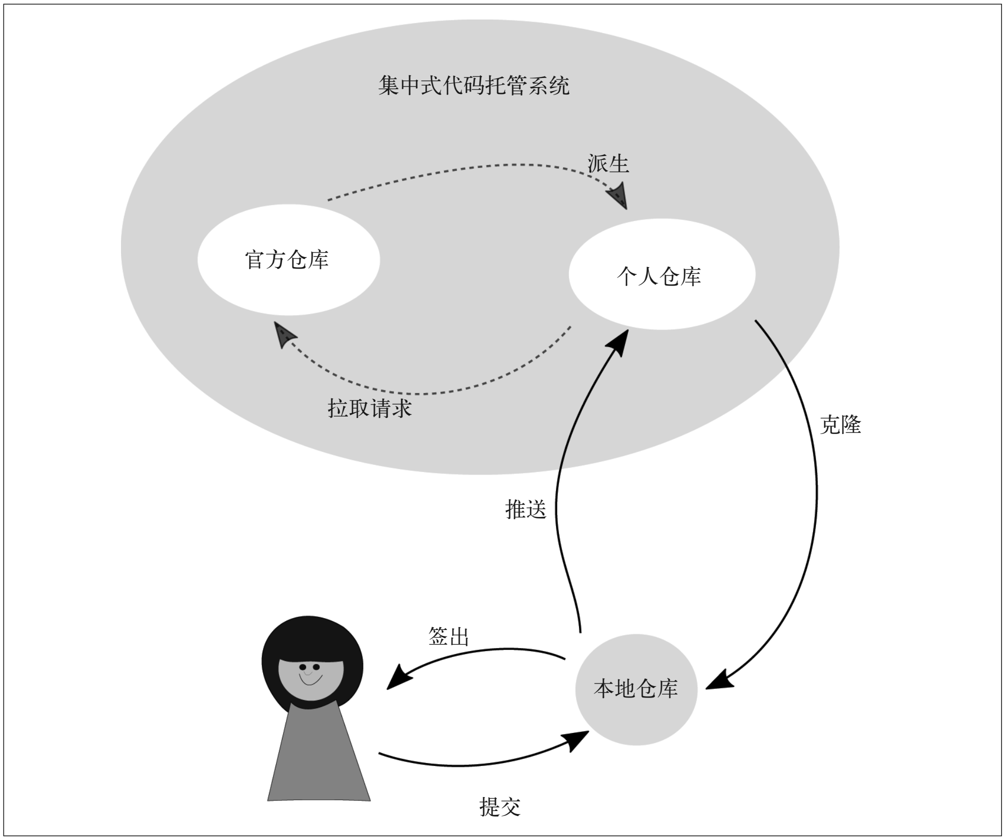


图 2-5：创建一个克隆仓库

当 GitHub 创建一个*派生*的仓库时，它等同于使用 Git 命令 `clone` 来创建一个仓库的副本。一旦你创建了一个派生仓库，你就可以在 GitHub 的网页上直接将更改应用于仓库，但对于比较复杂的更改来说，这不是一个好习惯。相比之下，你更有可能会再克隆一次仓库，而这次是从派生出来的仓库克隆到本地的工作区。这个做法高效地建立了从一个仓库副本到另一个的克隆链。保持所有仓库同步有些费事，但是，与直接操作补丁相比，需要记忆的命令会少很多。正所谓“有得有失”。

架构相同的仓库应该比分立的仓库更易于使用，因为你使用封装软件更加方便。除了让更新工作更为方便之外，封装软件使你能够更有效地控制谁能够提交工作并获得认可。

一般来说，克隆链中的第一个仓库应该只能被少数核心提交者改变，他们有权限向仓库中添加新的提交或合并分支。大多数项目的参与者将会从仓库的本地克隆开始工作。在这个本地的克隆仓库中，每个人都拥有完整的控制权限。他们可以添加新的分支、添加新的代码，并将工作推送到主仓库的公开克隆仓库，将提出的更改共享给别人。一旦工作被推送到了公共的克隆仓库，编写者可以请求社区对最新工作进行反馈。一旦工作被审查完成并由社区测试之后，编写者可以发起一个从公开的克隆仓库到主仓库的*合并请求*或*拉取请求*。

如果某人不打算让他们的工作贡献回到主仓库，他们可以跳过创建公开的克隆仓库这一步，直接将主仓库克隆至本地环境。如果你意识到有一些更改需要提交回到项目中，并且还做了一些不希望被共享的工作时，事情会变得有些麻烦。

不过，认识到你做的事能帮到别人并不总是那么容易的。比如，我正在使用一个开源的演示文稿框架 reveal.js（https://github.com/hakimel/reveal.js）为 OSCON 制作幻灯片。对你来说，你可能正在使用一个 WordPress 主题，或是一个前端框架，或是其他的起始模板。

以前在使用 reveal.js 制作幻灯片时，我觉得我不会在使用时升级 reveal.js 软件，若停下来升级，我会担心与上游项目的 Git 连接是否还存在。我在自己的仓库中翻遍了所有文件夹，我的工作才得以完成。一个自定义的主题完成了，我做了一些修改，我的仓库变成了一个派生仓库，和它的起点不再相连。（有开源项目经验的开发者这时会抓狂，因为他们已经意识到了我即将发现的事。）当我开始工作时，我发现幻灯片排版成讲义时出现了格式错误，我希望我的演讲者备注能显示在幻灯片的旁边（https://github.com/hakimel/reveal.js/pull/963），而不是堆积在下方。我在项目的 GitHub 页面上创建了一个 bug 报告，然后继续工作。有些人给了我一些重新排版的建议。啊！我知道怎么解决这个问题了。我想我的问题已经解决了，但其他人仍然有可能对我的解决方案感兴趣。现在我陷入了两难的境地，我创建自己的项目时并没有打算共享我的工作。

如果你提交了一个补丁，你或许可以只共享部分工作，但当你和并列贡献者一起工作时，你需要通过已有的仓库将你的工作共享回去。我自己的项目没有为上游的工作准备分支，因为我从来没想过要将我的工作共享回这个演示文稿框架。我开始创建一个新的仓库链。图 2-6 显示了我操作的顺序。在 GitHub 上，我创建了 reveal.js 主项目的一个派生仓库（https://github.com/emmajane/reveal.js）。然后我将派生的仓库克隆到本地。我在本地的克隆仓库中为我的更改创建了一个新的分支。之后我将这些更改从 OSCON 的幻灯片中（因为只有很少的幻灯片，所以我不介意打一个补丁，我只要用我习惯的复制粘贴工具即可）复制到克隆的演示文稿框架的仓库。当更改完成时，我将更改推送回 GitHub 上的远程仓库，并且创建拉取请求将我的更改并入项目。

reveal.js 仓库公开的克隆是必需的，因为我没有 reveal.js 仓库的写入权限。如果我有写入权限，我可以不必创建公开的克隆仓库，直接创建一份本地的克隆仓库。

2.2.3　共同维护的模型

最后，我们会介绍内部团队（和个人团队）中最常见的权限模型：共同维护。在这个模型中，团队成员之间有着天生的信任。我们假设代码在提交至主项目分支之前已经经过检查和确认，即开发者是受信任的。在这个模型中，在推送至项目共享的仓库之前，每个开发者在本地完成自己的工作。如图 2-7 所示，与内部团队协作时，我们的起点通常是单个共享的仓库，每个人都拥有对该仓库的共享写入权限。

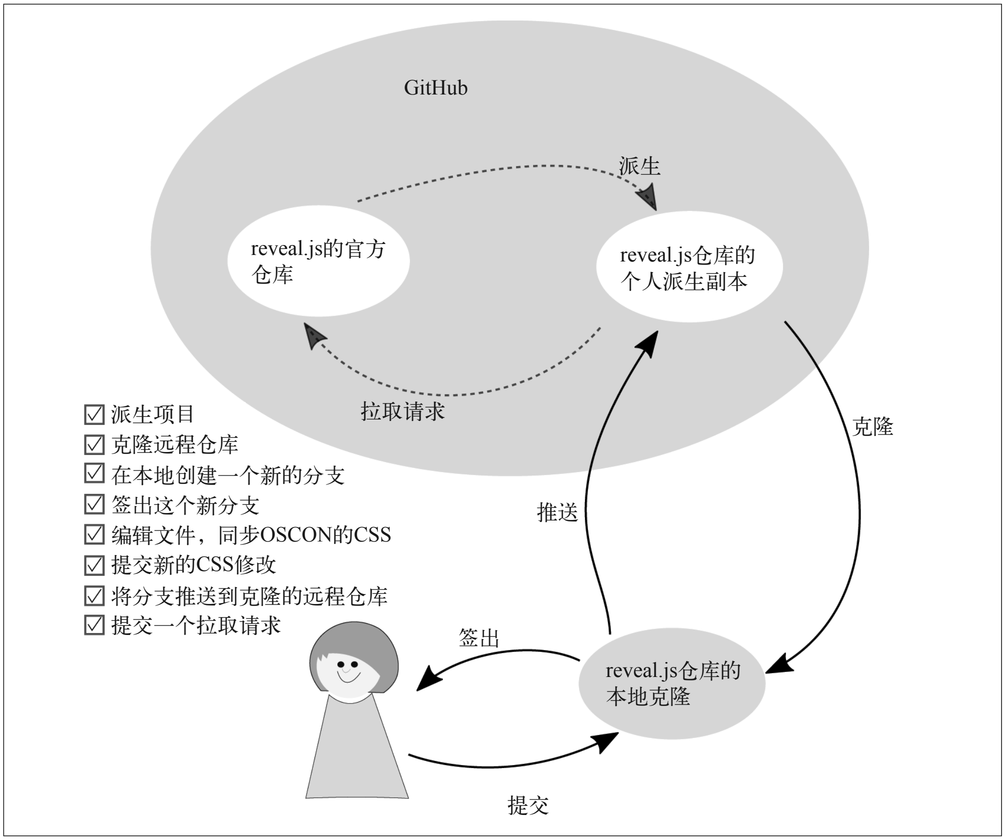


图 2-6：并列仓库之间的项目更改流向

Git 不提供权限控制，而是依赖其他系统来授权或阻止对项目的写入访问。如果你想要阻止别人将代码上传至共享仓库，需要使用托管系统的访问控制才能实现。如果你使用的托管平台并非 Git，那么访问可能由 SSH 账户管理。

除此之外，与 Subversion 不同，Git 不允许你锁定特定的分支。如果不借助额外的软件，团队成员默认在充分测试之前不会将更改提交到特定分支。Bitbucket（第 11 章）和 GitLab（第 12 章）提供了分支独立的访问限制。如果你倾向于使用一个更加轻量的系统，可以看一看 Gitolite（http://gitolite.com/gitolite/index.html）。

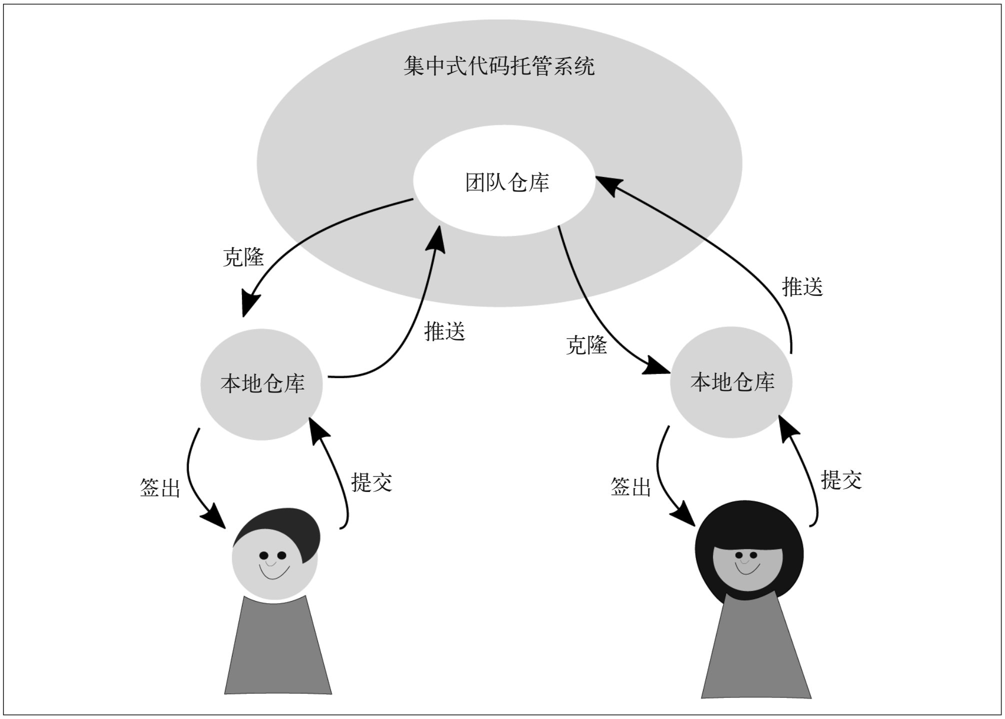


图 2-7：团队中每个成员都拥有从本地仓库向中央仓库写入的权限

2.2.4　自定义访问模型

除了这些策略之外，团队还可以为一个项目选择多个访问模型。官方仓库的代码提交有着严格规定的项目，这个模型将会特别有用。确实，大多数开源项目对不同的贡献者提供了不同的访问权限。

一个常见的工作流如下所示。

- 一个官方项目仓库，只有极少数人可以向该仓库提交代码。在开源项目中，这个角色即项目维护者；在闭源或合作的项目中，这个角色可能是 QA 团队。
- 一份限制更为宽松的仓库内部副本，每个贡献者和项目团队都使用该副本进行集成。这个仓库或许会遵循共同维护的模型，作为代码审查流程中的一环，每个人都可以将他们的分支并入仓库，甚至在一个特定的起点上继续开发。
- 锁定到各个贡献者的单独创建的个人仓库。通常这些仓库会使用与官方仓库相同的代码托管系统，因为大多数现代代码托管系统都拥有易于集成的功能（通常称为“拉取请求”或“合并请求”）。

这种项目组织方式常见于拥有初级开发者、QA 团队或者也许有外部承包商的团队。

第 4 章将更深入地讲解常见的工作流。

2.3 小结

在本章中，你学到了授权和限制访问项目仓库的各种方式。

- 清晰地定义项目治理模型将有助于所有贡献者理解项目的所有权。
- 代码的版权通常属于作者，除非因为贡献者协议或作为雇用作品将版权重新分配给另一个法人。
- 限制分发及代码衍生品的规则由软件许可证决定。
- Git 只是一个简单的内容跟踪工具，它不提供开箱即用的控制机制。一些代码管理系统整合了 pre-commit 钩子，用来限制分支独立时的访问。
- 任意仓库的访问可以是受限的，也可以是开放的。提交至仓库的更改由补丁组成。在代码托管系统中，你可以使用图形化界面来管理补丁提交流程。

仓库中的权限结构已经搭建完成，我们接下来将会探索如何分隔仓库，让你正在进行的工作和已经完成的工作都可以在团队成员之间共享。

第 3 章

分支策略

在版本控制中，分支用于隔离一块代码上产生的不同想法。分支总是起源于代码库中的某个特定节点。在第 2 章中我们谈到了派生和克隆仓库。分支类似于新工作开始时仓库内部的分隔。创建分支时你可能想要将工作贡献回去，也可能想要隔离不同的工作。分支本身并不关心跟踪的是什么更改！它们的职责仅仅是跟踪更改。

使用什么分支策略取决于你的发布管理流程。分支允许你更改项目工作目录中可见的文件，并且一次只会有一个分支活跃。大多数分支策略是根据粗粒度的想法分隔项目中的工作。这个想法可以是软件的版本，比如，版本 1、版本 2、版本 3。这些软件版本或许能使你联想到分支上正在进行的工作。根据它们代表的功能名称，这些想法被分隔在不同分支。这些想法可能是一个 bug 修复或是一个新的功能，但它们也可以表示一个完整的小想法。

本章概述了以下内容。

- 如何为你的团队选择一种分支约定
- 主线开发
- 功能分支部署
- 状态分支
- 计划部署

分支的使用方式没有任何限制。这既是好事，又是坏事。一些人为的限制（约定）将会帮助你考虑团队的无限可能。

3.1 理解分支

无需研究 Git 内部的工作原理，基本理解了分支的概念就能够帮助你挑选并应用本章概述的策略。

每个 Git 仓库都包含一个提交库。这些提交通过元数据相互链接，每个提交包含一个指向自身父提交的引用。对于合并提交（merge commit）而言，它可能会引用不止一个父提交。我通常把分支比作珠串，每个提交代表串上的一颗珠子。这个比喻不是那么严谨，不过很好地说明了我们的意图。Git 中的分支实际上是一个指向某个特定提交的命名指针。（给你一根魔棒，边说名字边敲一颗珠子。你创建了一个命名分支。）当你签出分支时，你将提交对象（由指针标识）中储存的数据复制到你的工作目录。在工作被复制到工作目录后，你可以进行任何操作（新增、编辑和删除文件），并且将更改作为一个新的提交对象储存到本地仓库。命名指针会自动更新并指向你刚创建的提交对象，同时你的分支也将更新。

你创建的任何提交对象都是本地的，并且只属于你，直到你选择主动将它们共享到一个远程仓库。这一点 Git 和集中式版本控制模型完全不同，后者提交更改时工作会被自动上传。对于一些可能出现的冲突，只需记住每个开发者在自己的仓库中都有一根魔棒，拥有全部能力。

开发者创造了关于命名分支和使用分支的约定来避免冲突。这些约定帮助开发者决定在什么情况下允许工作出现偏离（创建新的分支），以及什么时候合并（合并两个或多个分支中的提交对象）。一般来说，一种约定包含了两类分支：长期活跃的公开分支；短暂的私有分支。长期活跃的分支扮演了代码中介的角色，并入大量开发者的贡献。短暂的分支的作用是隔离一个新想法的开发过程。这些新想法可以是一个 bug 修复、新增功能或实验性的重构。这完全取决于你！

当你和其他人共享分支时，你可以继续在你的分支副本上添加提交对象；但是，既然分支已经被共享，可能其他人也在*他们*的分支副本上添加了提交对象。当你下次想要同步这两个分支时，Git 作为纯粹的内容跟踪者，会转而借助你的经验来将这两份提交对象合并到一个共享的历史。自动化流程中的这个中断称为*合并冲突*（merge confict）。我承认它听上去有些吓人。你的任务是*解决冲突*并为有问题的工作选择最合适的共享历史。

在 3.4 节中，你将会学到保持分支最新的策略。在第 7 章中，你将学到一些实用的命令。第 7 章同时介绍了如何解决冲突。不过，让我们先来看一看开发者在 Git 中维护工作时，一些最常用的分支命名策略。

3.2　挑选约定

约定是大家认同如何做一件事的标准。作为开发者，约定让我们能够快速了解一个软件项目是如何运转的，并且在整合自己的工作时不影响其他团队成员的流程。形成文档的约定让新参与者更加容易上手，而其他团队成员现在只需花费更少的工作时间来帮助新人。

为团队挑选合适的分支策略，需要成员们共同讨论希望如何发布你们的工作成果。（从现在开始，我将使用“软件”来指代你的项目，尽管 Git 也可以用于其他事情，比如写书！）对于网站来说，你或许会希望每天安排一次发布，但对于可下载的软件产品，你或许会希望每个月、每个季度甚至每半年安排一次发布。一旦确定如何发布软件以及是否有审计或跟踪的需求，你就可以挑选一个最合适的分支策略。

如果你已经了解工作方式，那么先花些时间整理一下你的需求，然后再深入研究细节并选择一个最合适的分支策略。如果你不确定你们的系统会是什么样的，第 4 章将会给你一些关于如何组织团队内部行为的建议。

只要你的团队将手头的工作形成文档即可，就不需要再制定硬性规定。事实上，如果你查阅几个开源项目的仓库，就会看到没有一成不变的做法。我推荐使用 GitHub 镜像来比较 Drupal（https://github.com/drupal/drupal）、Git（https://github.com/git/git） 和 Sass（https://github.com/sass/sass）使用的分支策略。这三个非常流行的项目使用了截然不同的分支策略。

不会有版本控制警察站在你门前告诉你做错了，况且你也总是能找到另一个与你风格相近的团队。但如果你刚接触版本控制，或者你的团队苦于寻找平稳进展的办法，选择本章介绍的约定或许能够帮助你。

3.3 几种约定

对于软件项目来说，团队通常有两种方式可供选择：要么选择“优先合并”，要么查对完成的工作并一次性发布。介于这两个极端之间有很多不同的工作方式。

本节概述如今开发团队最常使用的一些策略。你可以选择完整地采取其中一种策略，也可以根据你的需求来调整。不管你选择什么策略，记得把你的决定形成文档。

3.3.1 主线分支开发

主线分支开发是最容易理解的分支策略。在这个策略中只有少量分支。开发者总是将他们的工作提交到一个中央分支，这个分支随时可部署。换句话说，项目的主分支应该只包含经过测试的工作，并且绝不应该被破坏。

我通常独自参与一些不怎么需要使用版本控制的业余小项目，比如为杂志写稿。如图 3-1 所示，在这些情况下，我把所有工作提交到默认分支（在 Git 中称为 master）。如果我同时进行着两个不相关的想法，我可能会偷懒一起提交，或者保存（`stash`）一些工作供日后使用。对于这些简单的项目来说，即使不将想法分隔在不同分支，也能高效地工作。

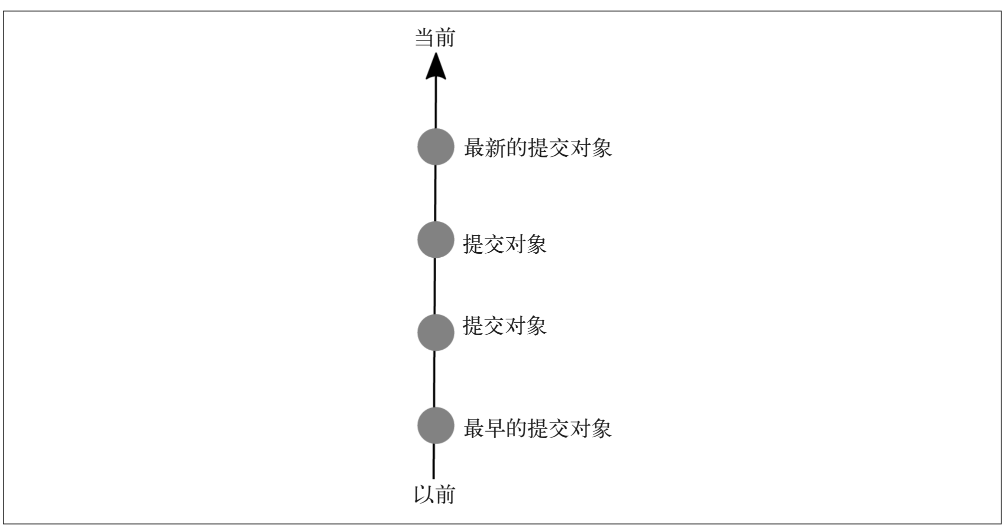


图 3-1：主线分支开发：将所有提交保存在一个分支

阅读链式图

图上的每个圆圈代表 Git 仓库中一个可溯及的提交。“链式”提交图的学名叫作有向无环图（DAG）。我敢保证没有人会考你是否记住了这个名称。但你若想进一步探讨，这会是一个很有用的术语。

随着项目成熟，会不断出现更多需要考虑的地方，跟踪想法也会变得越来越困难。如果我想要尝试项目未来的方向，而这些想法还不如当前其他工作那么完善时，我就会开始添加新的分支。我可能会扩大团队，像图 3-2 中一样，让一两个评审者负责他们自己的独立分支。当项目（以及团队成员）逐渐成长时，分支的数量也会逐渐增长。但这些分支不会同时保持活跃。正如《金发姑娘和三只熊》中的故事，你的团队可能会觉得一定数量的分支类型“不多不少正好”。每个工作单元［或者冲刺（sprint）］中的分支数量可能会存在手风琴效应。起初，开发者都在做自己的工作，而分支的数量随之增长。后来，当每个开发者完成自己的工作并将更改与其他人整合后，分支数量又降了下去。

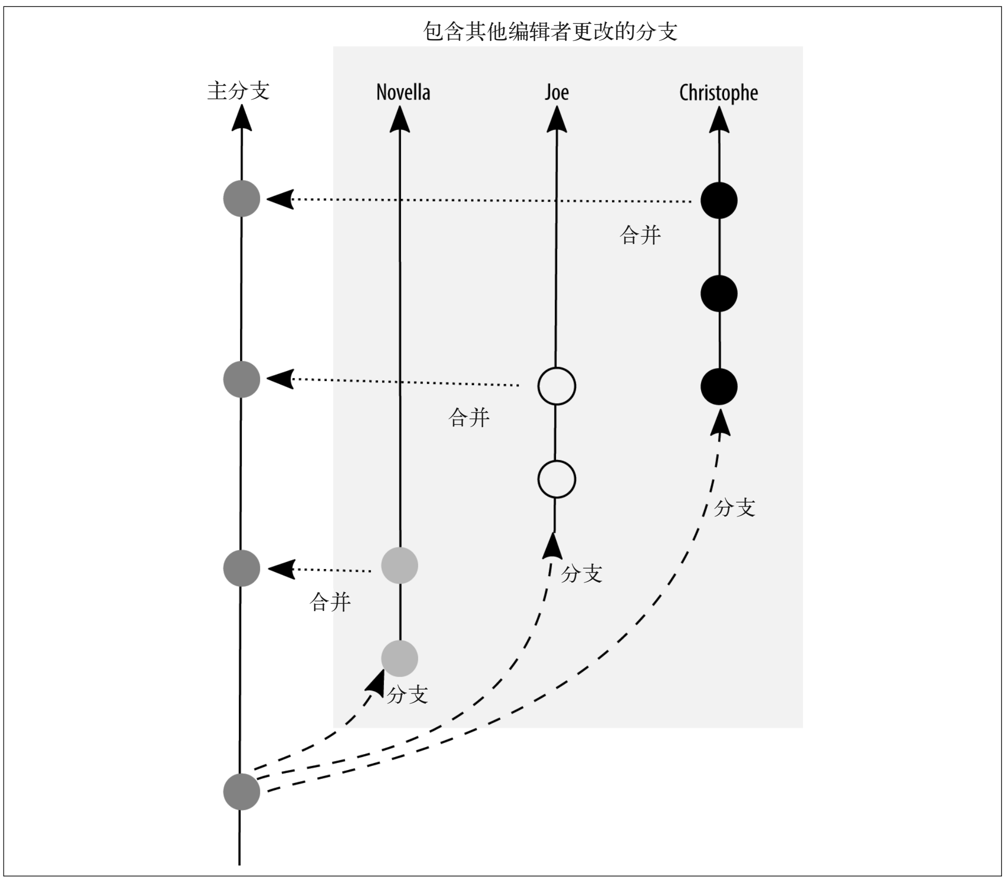


图 3-2：使用分支的主线开发：分支隔离了多人贡献的工作

使用单一工作分支的做法被自动化构建过程的团队大量应用。

为持续部署的团队提供的建议

持续集成是指所有开发者每天多次将自己的工作并入项目主线的实践。持续交付是指自动进行从开发者的本地工作站到服务器的中间步骤（但不是通过自动化的流程来部署）的实践。持续部署的自动化最为完备，因为所有代码通过一系列测试后，直接进入产品服务器。

将工作定期并入中央分支，但偶尔进行部署，这样的做法或许适合你的团队。在开始收集工作后，你需要将本地的工作和产品服务器上的工作区分开来。如果所有代码都是可部署的，那么添加一个小的修复然后立即发布不是什么难事。但如果你提交到仓库中的更改只进行了一半呢？因此，我们不再使用纯粹的持续部署策略，而是使用多个分支的计划部署策略。

使用鼓励持续集成工作的分支策略有以下优点。

- 整个项目中不会出现很多分支。因此，一项更改的消失不会造成明显的困惑。
- 并入代码库的提交相对较小。如果发现了问题，能够相对更快地撤销错误。
- 紧急修复会变得更少，因为主分支中的代码都是达到部署标准的。部署对于开发者往往都是压力重重的，他们紧张地看着代码进入产品然后等待用户的反馈。经过频繁的小幅更新，这个流程慢慢成熟并最终自动化，终端用户几乎感受不到。

使用这个策略也有下面这些缺点。

- 这个策略的前提是主分支中只包含随时可部署的代码。如果你的团队缺乏测试设施，臆想新的代码不会有问题就过于冒险了，尤其是在项目随着时间变得越来越复杂的时候。
- *部署*这个概念更适合指自动装载到用户设备（如网站）的代码。对于必须手动下载和安装的软件来说，这个术语不是那么合适。尽管修复问题的更新总是受欢迎的，但如果让我每天在手机上下载并重新安装一个应用，我也会感到不悦。
- 开发者验证产品代码的方法之一是将功能隐藏在一个开关后。据说 Facebook、Flickr 和 Etsy 都使用了这个方法。不过，这个方法存在的潜在风险是，隐藏在开关后的代码可能会被废弃，因为被隐藏而未被及时移除的代码会导致技术债务堆积如山。

不幸的是，介绍如何搭建持续部署的基础设施超出了本书的范围，因为这依赖于你所使用的语言（不同语言拥有自己不同的测试库）和部署工具。如果你希望了解有关理念的更多内容，Jez Humble 和 David Farley 的《持续交付：发布可靠软件的系统方法》[1] 是一本很好的入门书。

3.3.2　功能分支部署

为了克服刚才提到的单分支策略的限制，你可以引入两种类型的分支：功能分支和集成分支。严格地说，它们不是两种类型的分支，而是依据每个分支的职能来区分的约定。

注 1：此书已由人民邮电出版社出版。——编者注

在使用功能分支的部署策略中，所有新的工作都在一个功能分支上完成，这个分支小到恰好能够容纳一个完整的想法。这些分支通过一个集成分支与其他开发者完成的工作保持同步。等到软件发布前，构建管理员可以挑选将哪些功能集成到这个版本，并为部署创建一个新的集成分支。如图 3-3 所示，一次构建不一定会包含上次构建之后所有完成的工作。

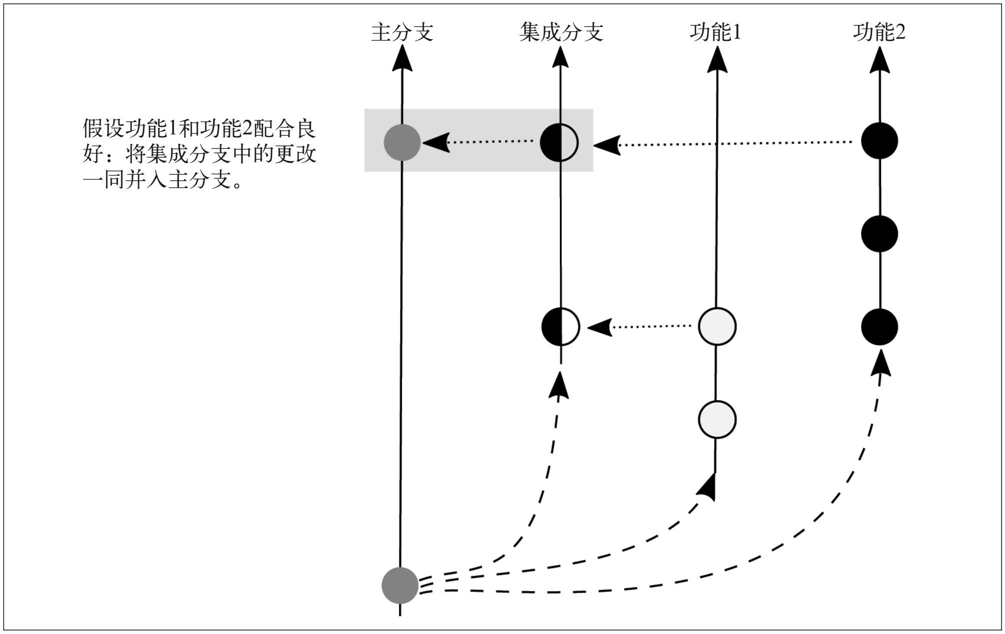


图 3-3：功能分支部署：功能分支之间通过集成分支保持同步

添加多个功能分支和一个集成分支后，你可以保证代码一直是可部署的，但在部署代码前也会停顿一下。这个模型最流行的描述是 Adam Dymitruk（http://dymitruk.com/blog/2012/02/05/branch-per-feature/）提出的，其更早一些的描述由 Scott Chacon 提出并被命名为 GitHub 流程（http://scottchacon.com/2011/08/31/github-flow.html）。经过了几次微调（https://twitter.com/emmajanehw/status/519814560539480064）后，这个流程如今仍然被 GitHub 使用。

在 GitHub 流程的分支模型中，任何 master 分支上的代码都是可部署的。当编写新的代码时，GitHub 流程要求开发者创建一个语义化命名的分支，并将他们的工作定期提交到这个分支。这个分支与 master 同步，并定期推送到共享仓库中对应的分支，使得其他人可以看到哪些功能正在进行。当开发者认为工作已经完成，或者需要其他人的帮助时，他们会在 master 分支上发起一个拉取请求。工单系统将会出现关于工作方案的讨论。

到目前为止，GitHub 流程和 Dymitruk 模型几乎是相同的。它们的区别在于部署方式。在 Dymitruk 模型中，创建一个构建前需要挑选即将集成的功能。在 GitHub 流程模型中，在合并请求被接受后，即可从功能分支上立即部署工作，在这个意义上，这种策略更接近于主线开发。早先，GitHub 将自己的功能分支并入 master 分支，然后再部署 master 分支。现在，我们部署完功能分支，如果上面没有错误，它将被并入 master 分支，如图 3-4 所示。也就是说，如果功能分支存在问题，我们可以立即重新部署 master，因为这个分支永远处于工作状态。

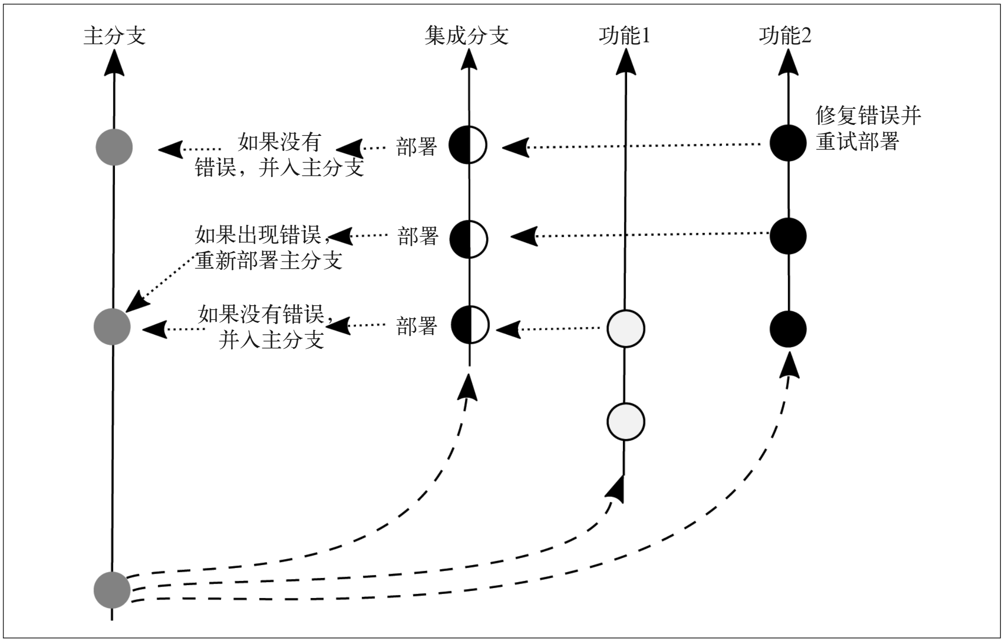


图 3-4：GitHub 流程：功能分支在审查后进行部署，然后被并入到 master 分支中

使用功能分支部署策略有以下两个优点。

- 类似于主线开发，我们力图让代码快速持续部署。
- 区别于主线开发，我们提供一个可选的构建步骤。当使用构建步骤时会出现一个选项，用于选择应该将哪些功能并入到 master 分支中进行部署。

同样，使用功能分支部署策略有以下三个缺点。

- 如果代码存放在功能分支上，而不是立刻发布到 master 分支，开发者需要进行额外的维护，保证在将代码并入部署分支前，功能都保持最新。
- 分支的语义化命名对于熟悉系统的开发者有帮助，但如果进行中的功能很多，新参与者会对这些行话感到困惑。
- 开发者需要定期清理那些已经并入 master 分支的遗留分支。这不算麻烦，但与在单一的 master 分支上工作，需要这个额外的步骤。

功能分支策略提供了一个介于主线开发和计划部署之间的过渡状态。在某种程度上，计划部署扩展了这个策略，但使用了特殊的命名约定。

3.3.3　状态分支

和之前提到的策略不同，状态分支策略引入了分支位置或快照的概念。我们使用的部署图通常过于简化，告诉我们代码从一个环境迁移到另一个环境（见图 3-5），但事情并没有这么简单。图 3-6 显示代码从一个分支合并到另一个分支，且其中每个分支都被部署到一个特定的环境中。（我们会在以后讨论带标签的发布，别心急。）如图 3-6 所示，我们

使用的分支名称和部署环境的名称通常不同。（master 是什么意思？这是用于产品发布的分支吗？还是开发的分支？你确定吗？）这个策略称为 GitLab 流程（https://about.gitlab.com/2014/09/29/gitlab-flow/）模型。

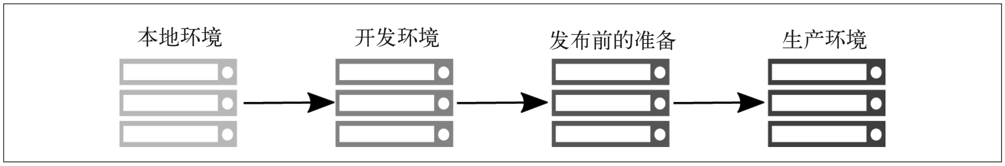


图 3-5：部署的谎言：代码其实并没有从本地服务器移动到产品服务器

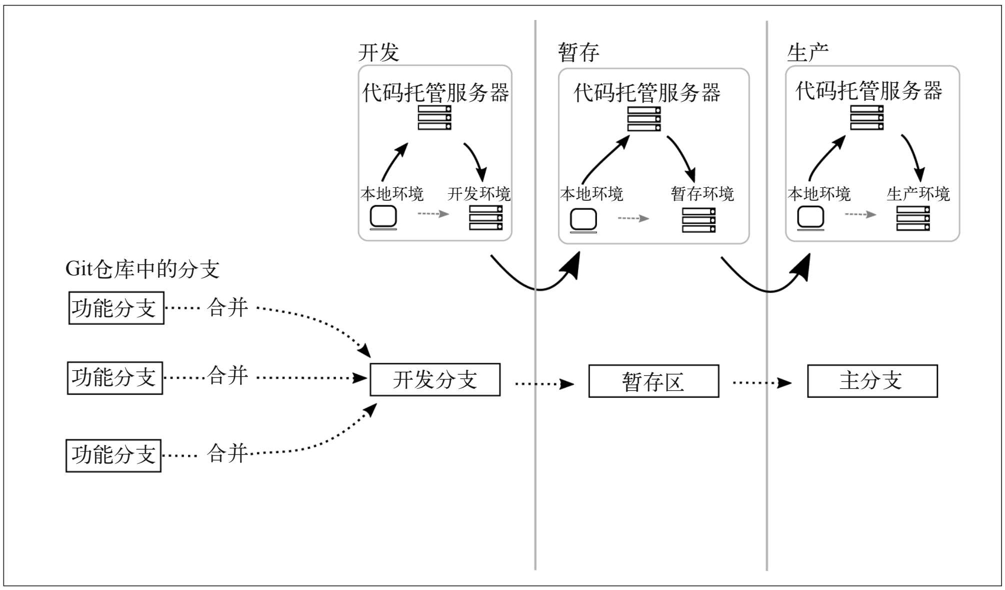


图 3-6：真实的部署过程使用了集中式的代码托管系统

通过分支的命名约定，GitLab 流程使我们明白什么代码用于什么环境，以及在合并提交前需要满足哪些条件。比如，你一定不希望将未经测试的代码并入一个叫作 production 的分支。或者，如果你要将代码交付给“外界”，GitLab 流程会建议你使用预发分支。理想情况下，这些预发分支应该遵守语义化版本命名（http://semver.org/）约定，尽管 GitLab 没有明确要求这一点。

在语义化版本命名中，如何增加版本号

在语义化版本命名中，发布的版本号总是应该遵循这样的格式：MAJOR.MINOR.PATCH。第一个数字（MAJOR，主版本）应该在进行了无法向后兼容的 API 修改时增加。第二个数字（MINOR，次版本）应该在增加新功能但是不会打破已有的功能（也就是*向后兼容*）时增加。第三个数字（PATCH，修订版本）应该在进行了向后兼容的 bug 修复时增加。

Git 项目使用的分支命名约定（https://www.kernel.org/pub/software/scm/git/docs/gitworkflows.html）是状态分支策略的一个很有意思的变种。它拥有以下四种专门命名的集成分支。

- maint
  这个分支包含了 Git 最近一次稳定发布的代码以及小数点版本的额外提交（维护）。
- master
  这个分支包含应该进入下一次发布的提交。
- next
  这个分支用于测试一些主题在进入 master 分支后的稳定性。
- pu
  这个建议的更新（proposed updates）分支包含尚未准备好合并的提交。

分支之间的关系就如同一个堆叠的金字塔一样。每一个“底层”的分支包含“上层”分支中没有的提交。如图 3-7 所示，maint 拥有最少的提交，而 pu 拥有最多的提交。一旦代码进入审查流程，它将被并入下一个集成分支，离进入官方发布又进了一步。

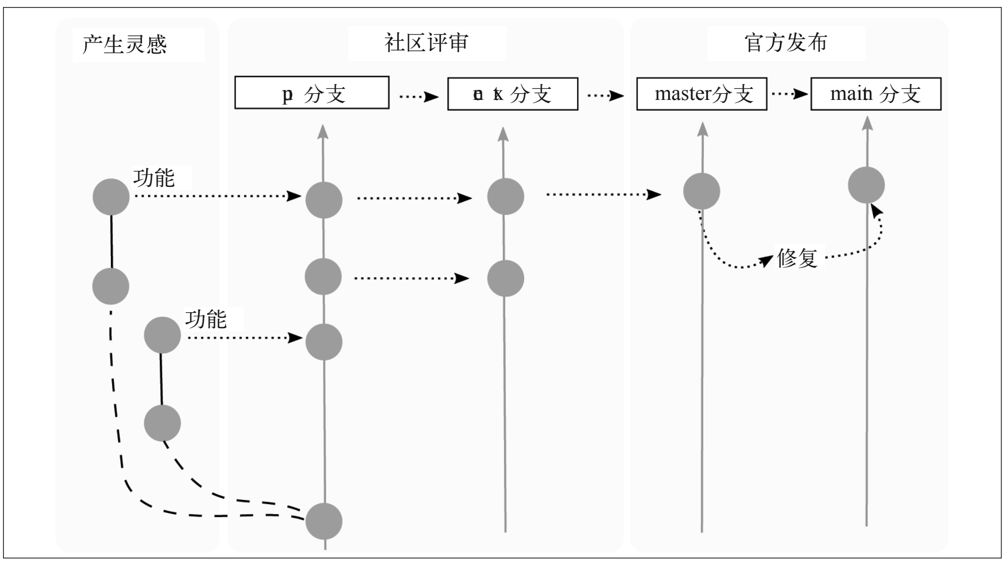


图 3-7：Git 项目使用的集成分支

使用状态分支策略具有以下两个优点。

- 分支名称与手头工作的上下文密切相关。
- 无需猜测就能明白每个分支的目的，在合并自己的工作时更容易选择正确的分支。

同时，使用状态分支策略也有以下两个缺点。

- 在没有指导的情况下，并非总是能很容易找到从哪开始新建一个分支。
- 由于分支名与团队的上下文密切相关，在项目中保持一致可能会更困难，使得新人上手时更加困惑。

在我自己的项目中，我通常选择这个风格的分支策略。我喜欢使用我容易理解的词汇，而不是其他团队中的其他人熟悉的词语。如果你对细节十分在意，那么请保持一致！除非你倾向于使用自己的词汇。

3.3.4 计划部署

如果你没有一个完全自动化的测试集，却必须计划一个部署，那么计划部署分支是最适合你的策略。或许是因为你需要遵循部署窗口期（比如，从不在下午 4 点后部署，也从不在周五部署），或许是因为要通过额外的机构审查（比如，在将 iOS 应用部署到 App Store 之前）。只要有人介入审查流程，或其他人对你的部署过程施加了任何限制，你的部署就不可避免地会出现延期，在等待时你需要想办法中断你的工作。

通过使用不同类型的分支策略，我们不断增加仓库内分支的复杂度。最开始我们只有一个分支，然后加上了功能和集成分支。在计划部署中，我们也要这么做。但是，计划部署的分支模式可能会渐渐变得非常复杂。复杂度应该逐渐上升，而且仅在必要时增加。

在本节中，我将会带你了解如何在团队中实施 GitFlow 分支策略，并了解其演化历程。GitFlow 是计划部署策略最流行的实现，最初由 Vincent Driessen（http://nvie.com/posts/a-successful-git-branching-model/）提出。它被世界上无数的团队用于构建软件项目，其最终形式可能看上去非常复杂。但幸运的是，软件项目的构建是循序渐进的，不是一蹴而就的。如果 GitFlow 中某些部分与你的团队无关，你可以在你的项目中忽略它们。

我们一起来仔细了解这个模型。

最初你的软件项目只有一个分支，`develop`。在这个分支上，程序员创建与之偏离的分支并在上面添加他们的功能。图 3-8 显示，此时 GitFlow 的状态图与本章先前介绍的模型十分相似。在这里，我会使用广义的“功能”术语。在理想的团队协作中，功能将在你开始工作前在工单中进行描述，且分支名将会包含这个工单名。比如，如果你有一个工单“1234”，代表一个修复失效链接的 bug 报告。若使用 `[ticket_id]-[terse_title]` 的命名约定，那么你的分支名应该是 `1234-fixing_links`。

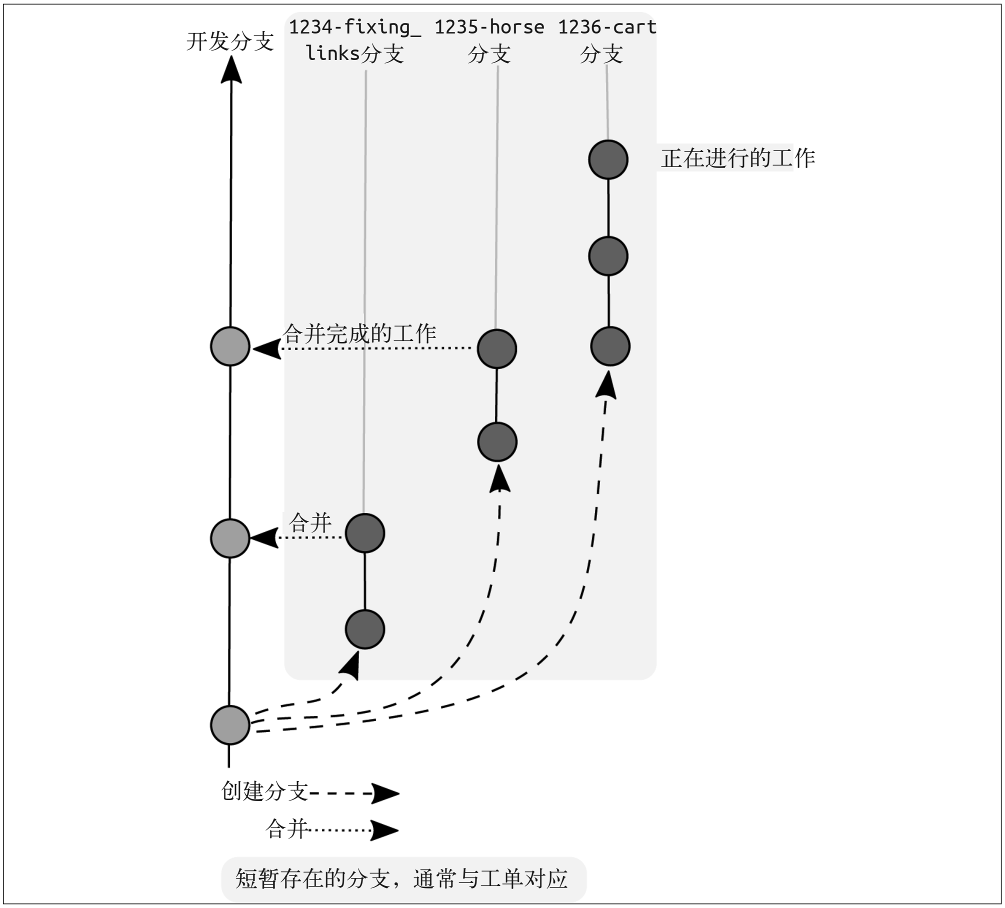


图 3-8：GitFlow 中使用的开发和功能分支

你的团队不断地工作、工作再工作，直到某一刻宣布“没有新的功能要做了”。我们通常将这个时间节点叫作*功能冻结*（feature freeze）。此时，会从开发分支上创建一个新的分支，如图 3-9 所示，只有 bug 修复能够提交到这个分支上。这些 bug 可能包括性能衰退、安全漏洞和一些其他问题。在更传统的瀑布流团队结构中，bug 修复的阶段应该由质量保证团队带领。在更敏捷的团队中，开发者从一系列分支到部署都一直跟进这些问题，甚至可能会负责测试其他人的工作。我们将会在第 8 章中更详细地讨论评审过程。

功能冻结时或许不是所有功能都完成了，所以仍然有工作被提交至 *develop* 分支。如果发现了 bug，这些 bug 同样需要被“向后”并入 *develop* 分支。图 3-10 显示了在代码并入两个不同方向时，我们看到的分支图。你的质量保证阶段越长，*develop* 分支和预发分支上越有可能出现同时进行的工作。

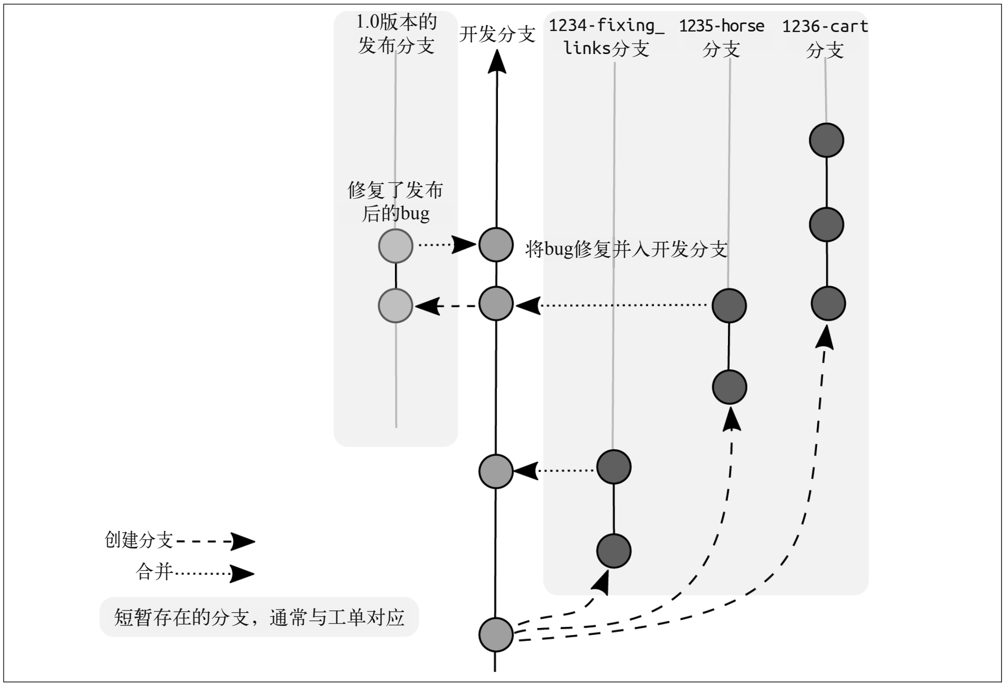


图 3-9：GitFlow 中的功能冻结；只允许 bug 修复

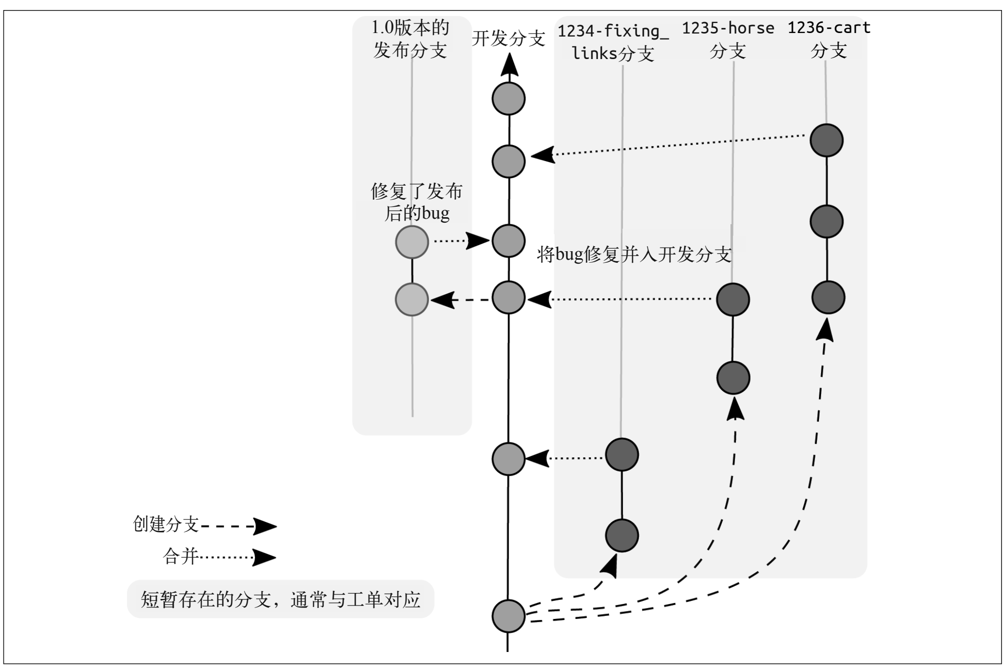


图 3-10：开发继续，但不会并入预发分支

经过一段时间的测试，所有 bug 会被宣告发现，剩下来的事情就是准备好部署。恭喜你！此时，所有经过了质量保证测试的代码都被提交到一个新的分支，master 上，这些代码被打上了软件当前版本号的标签（就像书签一样）。接着这个软件会如图 3-11 所示的那样进行部署。你的项目经理给你一个心形糖果，或者一个 GIF 动画，然后你就可以下班了。干得漂亮，队员们！（如果你的项目经理没有这么做，请把我的建议发给他们，我会帮你和他们谈谈。我们都是朋友，没什么大不了的。）

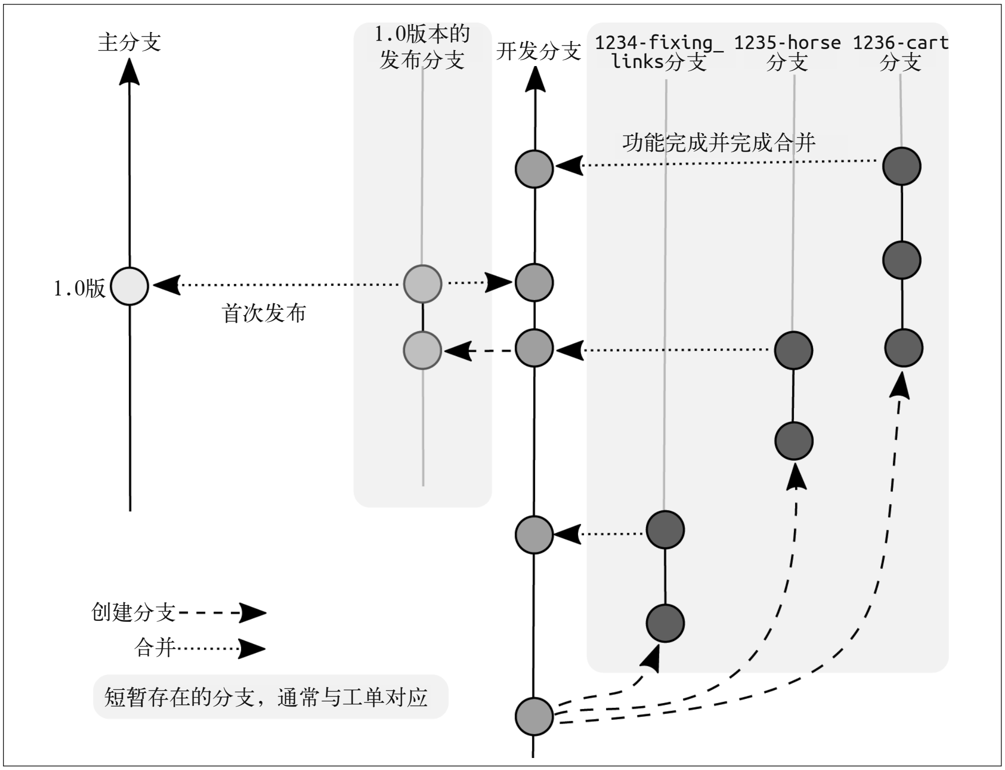


图 3-11：代码并入一个新的分支 master 并打上标签，从而软件得以发布

当然，在现实中，有些需要立即进行修复的 bug 将会潜入软件中。这些补丁（hotfix）十分关键，程序员应该在晚上回家前就修复这些 bug。它们通常是从产品分支拉出的一条新的分支，当补丁发布后，它们不包含任何上次官方发布后进行的额外工作，如图 3-12 所示。

在你的团队中定义“紧急”

有一次，一个与我合作过的开发者告诉我，如果一个 bug 必须要在他去酒吧喝杯啤酒之前进行修复，那么必须将这个 bug 标记为补丁。他的话从根本上改变了我如何理解那些标记为紧急的问题。我们重新校准了“紧急”的定义，最终减少了熬夜的次数。同样地，我曾经接触过的一个客户喜欢把工单标记为“非常重要，但不急着做”。你可以拿你的命名约定开玩笑，但确保记下了它们的含义，避免由于困扰造成的匆忙赶工。

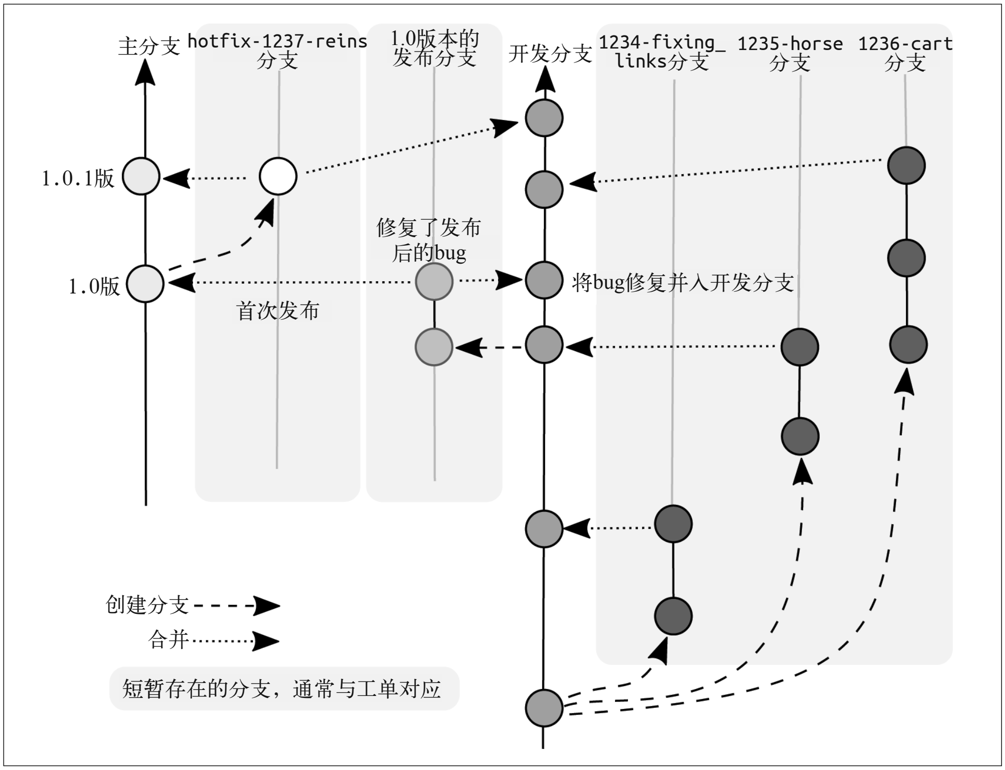


图 3-12：一个补丁完成后进入 master 分支，现在我们的发布标签是 1.0.1

由于需要在不同场所中继续不同的工作，我们渐渐构建了这些分支。你不需要一开始就建立所有分支。事实上，最好不要这么做，因为到最后你会发现需要维护的代码更多了。一旦你同时拥有了产品代码和开发代码，那么最终会在分支状态图中拥有一大堆活跃的节点，正如图 3-12 所示。对于新人来说这会是非常麻烦的，但对于伴随项目一路走来的开发者来说，这是一个自然的过程。如果你选择使用这个约定，先前有过相似经验的新来的开发者也会感到非常熟悉。

使用计划部署策略具有以下四个优点。

- 计划部署不要求你预先准备好全面的测试设施。
- 软件的构建过程通常伴随着开发、质量保证、生产这些阶段。也就是说，一旦软件开发者理解了分支约定中如何以及何时进行他们的日常任务，他们便会对 GitFlow 约定感到非常熟悉。
- 通过严格遵守约定，开发者总能明确自己应该从哪个分支开始工作。
- 这个模型同样适用于版本化管理的软件，如应用商店中下载的应用，这些应用不适合每隔几天部署一个新版本。

使用计划部署分支策略也有以下三个缺点。

- 对于刚接触到软件部署并且没有经历过整个开发流程的开发者，存在较多的认知负担。

- 如果开发者从错误的分支开始工作，要将所有工作恢复同步是件困难的事情。
- 它不如持续集成那么流行。

计划部署策略提供了最严格的约定，规范了代码应该如何在通过评审后移动。它通常在几乎没有自动化的代码评审中被使用，并且总是出现在没有应用自动化部署计划的某些项目中。工作收集到项目且尚未发布时，你将会至少见到本节中描述的一些特征。

3.4 更新分支

本章主要介绍用于隔离和合并几股工作的常见策略。这些策略集中于一个单一最佳路径的场景中，其中不同分支的工作神奇地与其他相关的工作保持同步。在分布式版本控制系统中，整合外部工作的方法与你选择的分支策略无关。更新分支时，你可以选择其中一种策略：合并（merge）或变基（rebase）。在深入了解这两种策略的差异前，我们先来简单地了解一下多个仓库之间的连接是如何维持的。

每个 Git 仓库都是一份自治的更改记录。仓库之间的连接通过一个远程的引用建立。这个应用允许开发者将远程仓库中所有提交对象复制到自己的本地仓库。远程连接通常用于至少共用部分历史的仓库。比如，第一次使用命令 `clone` 下载仓库，将会出现一份远程仓库及其提交对象的副本。

例如，你想要将你的工作添加到同事的分支上。你连接到他们的远程仓库，拉取分支并尝试添加你的工作。但你不能这么做！如果那是一个本地分支，你可以在分支顶端添加一些新的提交对象。但是，因为你希望更新的是一个*远程*分支，所以你无法将一个新的提交对象放置到你的仓库中那个分支的顶端，因为这个操作只能由远程仓库的所有者完成。与此相反，你必须先创建一个新的跟踪分支来储存你的更改。

> **一些跟踪分支是自动的**
>
> `clone` 命令将会默认创建一个名为 `master` 的跟踪分支，和与它同名的远程分支保持一致。

现在你有了一个可以添加新提交的分支的本地副本，一个不能添加提交的分支的引用副本，以及一个仍然存在于远程仓库的原分支。当你和你的同事分别在自己的仓库中进行更改时，这些分支将会不可避免地不再保持同步。`fetch` 命令将会更新分支的引用副本，下载所有新的提交。这个分支的可变跟踪副本，有多种更新方式。这是因为你现在将两个分支并入一个，这个操作在 Git 中对应了多种策略。有选择就会有分歧，会争论应该使用哪种方法。

通过远程引用更新跟踪分支的过程通常借助 `pull` 命令实现。然而，`pull` 是两个无关步骤的组合：`fetch` 和 `merge`，或者 `fetch` 和 `rebase`。`pull` 命令默认使用 `merge` 策略来更新本地分支，但是，通过添加 `--rebase` 参数，开发者可以选择通过 `rebase` 策略来更新自己的本地分支。

两种变基的方式

变基在更新一系列提交时有两种方式。第一种，在相关分支上整合新的工作（更新一个分支）时用于代替合并。第二种，通过增删修改分支上的提交历史中的特定提交来更改现有分支上的历史，达到使历史更加精简的目的。本节指的是前一种方式。

变基给人带来的印象是复杂和繁琐的。但从图形化的角度来说，变基是可读性最佳的策略。图 3-13 显示了从一个分支变基至另一个分支前后的变化。一般来说，我们将变基解释为在一个已有的时间线上重放已有的提交。这个比喻虽然在技术上有些出入，但很好地解释了合并和变基之间的差异。

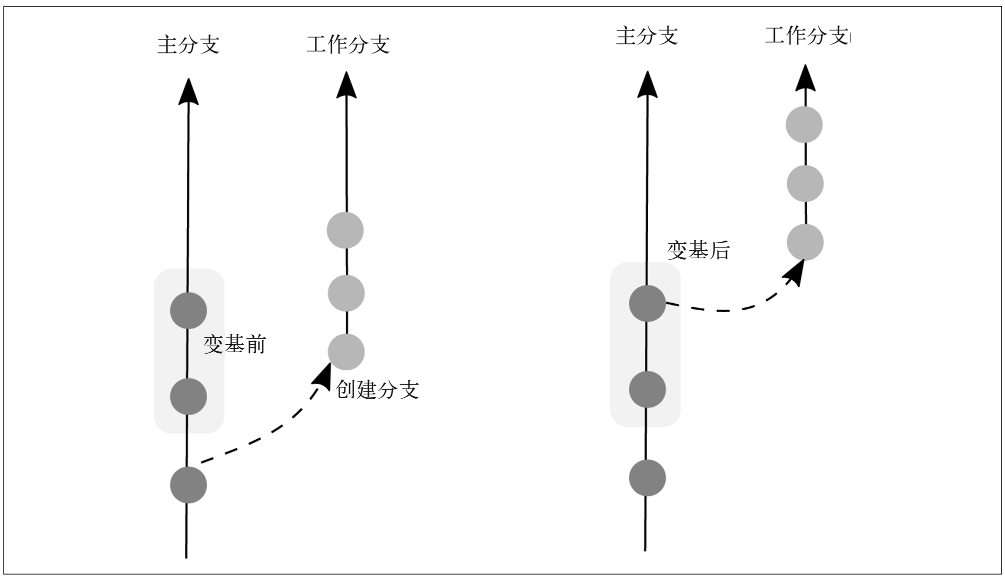


图 3-13：变基两个分支时只改变其中一个分支的历史，而另一个分支看上去没有变化

`rebase` 命令用于更新分支，而 `merge` 命令用于引入全新的工作。当 `merge` 命令用于快进（fast-forward）策略时，状态图和变基后的图形并无二致。快进策略只有在合并只包含并入分支中已有的提交时才能够使用。图 3-14 显示，快进合并的图形和变基一样整洁。

当两个分支上都有新的工作时，如果你希望合并这些工作，那么将会需要在一个新的提交中储存合并后的工作。你可以使用不同的合并策略，Git 会选择对你最合适的一种策略。如果你对不同的合并策略非常感兴趣，Git 介绍合并的帮助页面会告诉你章鱼式合并（两分支以上的合并）和递归式合并有哪些差别。要阅读文档，请运行 `git help merge` 命令。

不知道如何选择合并和变基？

使用快进合并或变基的两个分支所产生的状态图几乎是一样的。因此，选择哪种策略有些令人困惑。事实上，这种选择是如此困惑以至于一些团队选择混合使用这两个命令。如果你花费一些时间来理解在什么时候使用哪种策略，就将能够敏捷地在不同项目中选择不同的策略使用。合并还是变基（http://gitforteams.com/resources/merge-rebase.html）提供了一个决策树来帮助你认清应该使用其中哪种策略。

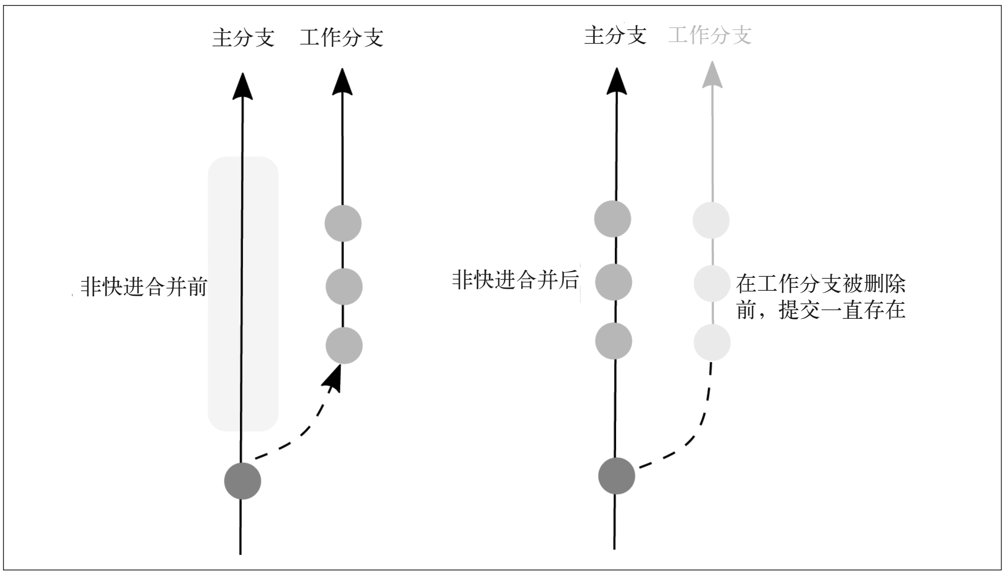


图 3-14：使用快进策略合并两个分支和使用变基一样整洁

如果你希望通过合并来让手头的工作保持同步，图形化的历史将会变得难以阅读，因为其中的连接变成了双向的。也就是说，历史在两个分支间突然转向，因为代码已经同步，而新的功能也发布到了主分支上。图 3-15 显示了合并是如何记录提交的来源的。如果你正在将一个功能分支并入项目的主开发分支，那么这会是你希望看到的。但如果你只希望阅读当前功能的历史，则会感到有些困惑，因为主开发分支将会进入你的历史图，而功能分支和集成分支合并后的连接都会被画在上面。

由于一些同步的问题，使用 Git 的开发者如果准备将他们的工作提交回项目，那么他们通常不会在跟踪分支上工作。取而代之，开发者将会创建这个分支的第四份副本（一份是跟踪分支的副本，跟踪分支是引用分支的副本，而引用分支是远程分支的副本）。不管使用哪种分支策略，跟踪分支可以映射到任何长期运行的分支（比如主分支或预发分支），且工作分支是一个功能、工单或补丁分支。

通过分支变基来保持同步使得图像简化，而历史变得更易于阅读。然而，变基也要付出一些代价，尤其是当你的分支副本中包含刚创建的提交对象时。为了变基一个包含独特提交

的分支，你必须将每个提交一个个重放至新分支的顶端，Git 为每个提交分配了一个新的标识符，因为每个提交都分配到了一个新的父节点。如果在分配到一个新的父节点提交之前共享至其他远程仓库，那么这可能会造成困惑。除了新的标识符以外，每当你重放一个提交时，都有可能出现合并冲突，有时解决冲突非常耗时。这有点像维护考勤表，只要你每天投入一点时间保持你的考勤表同步，就不会出现什么大问题。但如果你实在记不住每天在考勤表上做记录，那么回顾并补全之前的记录会是非常耗时的。通过变基策略维护一个同步的分支带来的奖励是一个易于阅读的分支历史。但它真的值得吗？如果 Git 新手还不怎么习惯解决合并冲突，那么他们会感到非常受挫。

留给你一个课后作业，和你的团队讨论以下哪一点更重要：易用性（选择合并来保持分支同步），还是易读的历史记录（选择编辑来保持分支同步）。

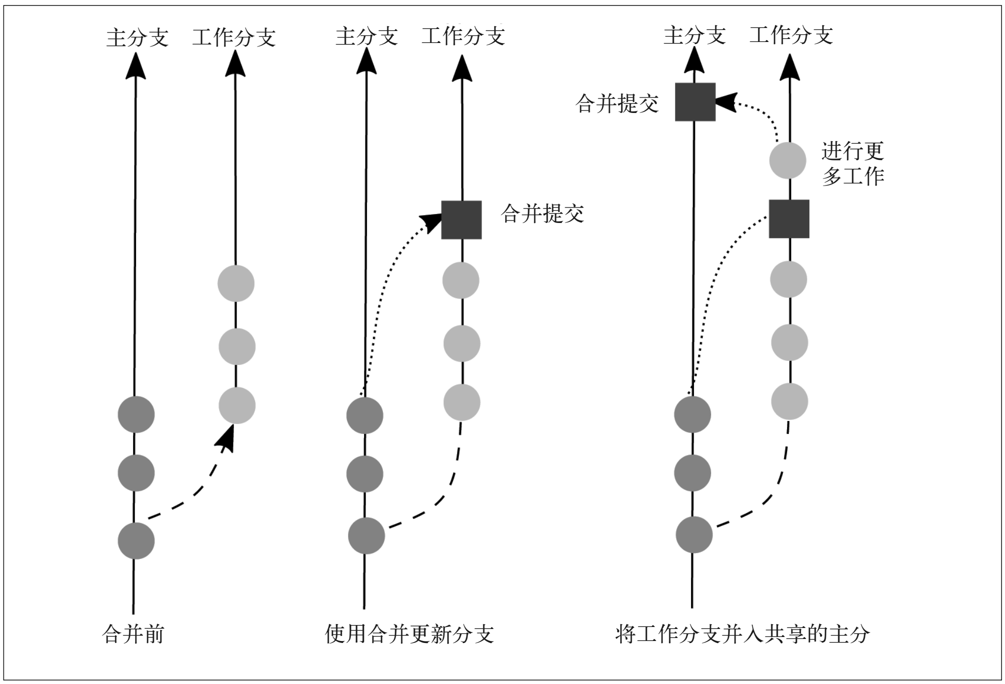


图 3-15：不使用快进策略合并两个分支

3.5 小结

如果你使用 Git 托管系统，如 GitHub、Bitbucket 或 GitLab，分支可能被用于分隔一个特定 bug 或功能工单对应的工作。根据你使用的分支策略，你的目标或许是让分支无限期地分隔，或者你或许希望经常将这些分别完成的工作合并到一个可部署的分支中。尽管所有信息都储存在仓库中，但在某一时刻只有一个分支可见。所以，如果你有两个想法正在同时进行，并且希望它们同时在服务器上出现，你将会需要将这两个分支并入一个常用的分支，来让它们同时可见。

本章介绍了使用 Git 时的几种分支策略，以及某些团队使用过的这些策略的一些变种，如下所列。

- 主线开发
- 功能分支部署
- 状态分支
- 计划部署

除了这些策略以外，你还将决定你的团队需要如何将各自的新工作并入共享的分支，并保持分支最新。对于新手团队，如何保持分支最新并不总是显而易见的。有两种策略可供选择：变基或合并。变基策略如果不定期执行，就会变得尤为麻烦，但它确实能使你的历史记录的图像更加整洁，从而易于浏览。通过使用合并来保持分支最新，浏览项目历史将会变得更困难。因此，如果你的工作起源不重要，可以选择其中任一策略，但如果你要经常浏览历史记录，变基会使未来的工作更容易（即使现在来看更耗时了）。

第 4 章

工作流

我喜欢和团队一起讨论并制定行动计划，便利贴和白板越多越好。在这个过程中，可能会有一些争吵和妥协，但最终大家能在最基本的流程上达成一致。每个人回到自己的桌前，思考自己要做的方向，并迅速发问“我应该怎么开始呢”。你给团队更多的提示来开始工作，他们就能将更多的精力花在攻克难题上。版本控制永远不应该是一个难题。

完成本章学习之后，你将能够创建涵盖以下内容的分步文档。

- 基本工作流
- 集成分支
- 发布计划
- 发布后的补丁

本章安排了一些精选的案例学习，介绍我在之前的团队中是如何高效地使用 Git 的。你将会注意到我对敏捷方法的强烈偏爱，尤其是本章中的 Scrum。这个协作流程与流行的工作流模型 GitFlow（http://nvie.com/posts/a-successful-git-branching-model/）配合良好。如果你已经对 GitFlow 非常熟悉，仍然应该阅读本章的第一节，了解如何建立并记录团队的工作过程。

4.1　初识工作流

在第 2 章中，你学到了治理模型。在第 3 章中，你学到了分支策略。使用 Git 的工作方式可能会很快变得非常复杂，而且复杂度越高，要记住所有事情就越困难。在你的团队中要建立约定来维护一致性，以帮助你快速理解你的代码历史。

在本节中，你将学到以下内容。

- 记录团队进展的基本工具

- 文档应该放在什么地方
- 什么类型的事情需要记录
- 工单系统中的样例状态

与你的团队聊聊他们想要怎样协作这个问题，这永远都不迟；改善你现有的流程也永远都不晚。如果你使用敏捷方法，那么应该已经安排了专门用于回顾会议的时间，或是使用持续改善（Kaizen）方法来评审开发过程。

4.1.1 记录工作过程

Git 作为一个无感知的软件，并不关心你是如何组织事物的。放轻松，如果你用错了分支名，或者把合并当成变基使用，Git 不会突然从你的电脑里跳出来，用手指着你，告诉你做错了（尽管我有时觉得 Git 如果这样做也挺好的）。你使用 Git 的方式完全取决于你自己。

最简单的保持一致的方法是遵循一组规则，或是一份清单。每次在一个新的地方工作时，应该记录下你的工作流。通过从模板开始（例 4-1）工作，能够确保现在看上去“显而易见”的细节，在新人到来甚至混乱到来时仍然是显而易见的。

例 4-1　模板工作流

```
项目经理:名字
开发站点:URL
开发站点上部署的分支:分支名
线上站点:53URL
线上站点上部署的分支:分支名
开始一个新的开发工单时,从哪个分支创建新的分支:分支名
开始一个新的补丁工单时,从哪个分支创建新的分支:分支名
更新工作时,使用:Git命令
评审后合并时,使用:Git命令
```

文档中包含的细节越多，队友之间的一致性就越强，回顾仓库中的历史记录也就越方便。

如果你的团队都在一起，那就坐下来勾勒一张图，描述代码中应该怎么区分权限。如果你在分布式团队中，也并不意味着就不能勾勒这张图了。而且你不需要变成一个插画家。有很多漂亮的图表软件能够帮你勾勒你的想法。我是 Balsamiq（https://balsamiq.com/）的狂热粉丝，常用它来画很简单的图。还有人推荐了 Pencil（http://pencil.evolus.vn/）、OmniGraffle（https://www.omnigroup.com/omnigraffle）、Dia（http://dia-installer.de/）和 Inkscape（http://www.inkscape.org/en/）。第 2 章中的图对于很多团队来说也是一个很好的出发点。这本书中所有的图都同时提供 SVG 和 Balsamiq 两种格式的文件。你可以前往“Git 团队协作”图表（https://github.com/gitforteams/diagrams）仓库下载。

4.1.2 记录编码的决定

我在整本书中都会提到工单或 issue[1] 的处理。开源软件项目的严谨性催生了非常多优秀的工

注 1：issue 本义表示一个问题。在 Git 托管系统中，Issues 是一个专门的板块，列出所有 bug 报告、功能请求和项目相关讨论。每一个 issue 为一条工单，而 Issues 就是一个工单系统。——译者注

作习惯，其中一个是使用 bug 跟踪工具来捕捉所有需求。对于开源项目而言，我在不同产品中使用了不同的跟踪工具，比如 Drupal 项目模块（https://drupal.org/project/project，它被亲切地称为 issue 队列）和一些通用的方案，如 GitHub（https://github.com/）。对于内部项目来说，我还用过 Pivotal Tracker（http://www.pivotaltracker.com/）、JIRA（https://www.atlassian.com/software/jira）、Redmine（http://www.redmine.org/）和 Unfuddle（https://unfuddle.com/）。

每个系统各有优缺点。我没有一个最喜欢的产品。从本质上来讲，这些系统允许你记录、跟踪即将完成的工作、需要完成的任务，以及与任何在质量保证测试中发现的后续问题相关的讨论。我无法想象如何与一个没有集中式工单跟踪工具来记录已完成工作的团队合作。

在一起工作的团队可以使用白板和便利贴来展示当前正在进行的工作。一些团队也用非常简单的表格来跟踪谁正在做哪个任务。或许对话和相关的资源（如图表、设计资源和线框图）可以存放在 Wiki 中，这样白板可以擦干净供下次对话使用。不论你使用什么系统，我鼓励你在易于阅读且方便搜索的系统中，至少记下决定做这些功能的原因。如果你没有把这些信息写下来，那么以后可能会苦苦猜测当初为什么作出了这些决定。

然而，如果在每次更改仓库时的提交说明中没有同时包含决定，那么团队会对使用的某个工单系统产生依赖。你的团队可能会认为对话是临时的，于是将结论记在了提交说明中，然后离开了对话。

这是一个平衡。诀窍是预测未来的对话，确保你的跟踪系统能够轻易地回答你的问题。或许你希望以后避免有开发者要求你在作出决定之后再重复一遍对话的情况。在这种情况下，你会希望工单系统能够展示双方（通过评论）讨论的进展、最终的结论和决定变成代码后对应的提交链接。或许你正在构建的软件遵从业界规定的监督，你需要证明这个软件经历了特定的评审过程。在这种情况下，你的软件仓库中包括了有单独的质量保证测试员签名的提交就足够了。

我不认为有哪个系统在跟踪软件开发上做得更好。很多系统既有优点，又有缺点。如果你使用了特定的流程管理哲学，其中提倡特定的任务工作流，那么你可能会觉得为该流程优化过的软件产品更易于使用。例如，看板（Kanban）是一个处理任务的具体例子。

大多数 Git 托管平台都自带一个基本的工单跟踪工具来帮你协调项目的开发。第三部分将更详细地介绍其中三个系统（Bitbucket、GitHub 和 GitLab）。

4.2　工单进展

即使你在没有固定期限的内部项目中工作，我仍然建议选取一小段时间，周而复始地进行。我个人的偏好是每周一个冲刺。对于内部项目来说，这些冲刺可以充当激励团队不断前进的任何最后期限。在每个冲刺最后，我建议组织一次内部展示，展现自己的工作。公开的展示会促使开发者对自己的工作负责。如果你的团队是分布式的，那可以在 Google Hangout（http://www.google.com/+/learnmore/hangouts/），或者适合更大团队的 GoToMeeting（http://www.gotomeeting.com/）上托管这些展示。

跟踪成员工作的项目方法都是由下面这些基本概念发展而来的。

- *尚未开始*

  在 Scrum 术语中，它称为*产品积压工作*（product backlog）。本质上，这是与当前工作（或者冲刺）无关的事情。开发者不应该从这个列表中挑工单来做。积压工作应该按优先级排序，提示团队哪些应该在下个工作冲刺中开始做。最近，我工作过的一个团队将它称为一堆“以后非常非常重要”的工单。

- *准备开工*

  当前工作周期中安排过优先级的工单。这些工单可能是积压工作表中积压的工单，也可能只是团队挑选出将要开始做的工单。你的团队可能希望将这个阶段分成几个子类别，如：准备开发、准备代码评审、准备测试、准备客户批准和准备部署。

- *进行中*

  开发者正在做的工单，或正在进行质量保证评审的工单。在更大的团队中，你可能希望同样细化这个类别。例如：制定中、开发中、测试中。

- *已完成*

  工作已经完成或被取消的工单。或许还会有后续的工单，但在经过代码评审、质量保证评审和客户评审之后，工单几乎不会被重启。

不要让项目经理过度细化分类

允许团队逐渐添加需要的状态。我在很多项目里见到过这种情况，项目经理在描述了所有可能状态的一大堆分类中做决定。这样的系统使用起来是繁琐不便的。（我是一个喜欢分类的经理！）开发者从来不喜欢记住要微调他们的工单状态，大多数情况下，这些工单的状态都是错误的，除非项目经理自己在状态之间移动工单。要理解开发者想要将宝贵的时间用在开发而不是更新时间表和微调工单更新上。那就从简单的做起，使用尽可能少的类别，当开发团队需要新的状态时再添加。

举一个稍有变化的例子。2014 年秋季我合作的团队有九个人，在项目过程中使用工单跟踪工具（一个相对小的项目，却是敏捷项目常见的团队大小）。工单跟踪工具包含以下状态的摘要栏。

- *准备齐全*

  这类工单准备开始做，且应该在本周完成。

- *进行中*

  这类工单正在活跃地进行中。

- *合并请求*

  代码已经完成，等待评审后并入主分支。

- *需要测试*

  代码已通过评审，并被推送到开发分支。准备在质量保证服务器上接受团队成员的测试。

- *已完成*

  工单已完成。这个状态也包括已关闭但未完成的工单（重复的任务、不再需要的功能）。

产品积压工作只是一堆没有分配状态的工单。

如果开发者遇到了解决不了的问题，他应该将这个工单重新分配给最有可能“解决”这个问题的人。和别人交换工单的习惯是文化的一部分，不可能适用于所有团队，但它对于分布式团队来说确实表现良好，因为在这样的团队中你无法做到拍拍别人的肩膀就能解决你的问题。

我比一般的开发者更喜欢分类系统。然而，增加复杂度是有后果的。复杂度的提升会增加人们决定将工单放入哪个类别的时间（这个工单应该是需要测试呢，还是合并请求）。它同时增加了开发者在工单系统中花费的时间，而不是在自己的代码编辑器上。它潜在地同时提升了与其他开发者沟通的时间，并减慢了真正做事的速度。你需要时刻提防这种现象，找到改进方案来优化你的流程。

选择你自己擅长的工单

在我工作过的团队中，开发者能够自行分配一些自己的工单，这么做的反馈良好。当然了，可能有一些工单需要某个人特殊的知识，但一个人发现自己可以完成这个工单与要求他去做什么的状态之间存在的差异是令人惊叹的。

与团队成员增加沟通永远都不嫌多。我不是指将自己的时间排满而没有组织会议。我的意思是真正地交流你们正在做的工作，有什么问题阻止了你完成这些任务。工单状态帮助你将沟通标准化，这会使保持同步更加容易，并确保团队中每个人养成每天确认一次工单状态的习惯。

4.3　基本工作流

这个基本工作流适合拥有一至二个可信开发者的团队。正如介绍中提到的，这是 GitFlow 流程的一个简化版本。它在不增加复杂度的情况下，整合了功能分支的工作流。因此，你会发现这个工作流对于拥有测试设施的开发团队，也就是致力于快速部署代码的团队来说表现良好。

关键特征包括以下这些。

- 治理模型：贡献者共同维护
- 集成合并：由原开发者执行
- 整合分支：*develop*

我使用这种工作流时的个人偏好接近于看板风格的系统，后者允许工单在一个工作板上流动，但是，我发现在时间固定的冲刺中使用 Scrum 方法，与外部利益干系人的沟通计划会更容易。在 Scrum 中，一组工单进入一个冲刺，我们的目标是在最后期限前将存在的工单数量降至 0。对于内部项目来说，Scrum 风格的冲刺计划可以充当保持团队不断进步的任意最后期限。

在每个冲刺后，我建议举行一个内部展示会，让团队展现他们的工作，如果他们卡在了某个特定问题上，可以向更多人求助。

这个工作流如下所示。

(1) 当你开始一个工单时，在工单跟踪工具中更新状态，比如将它标记为正在进行。这会通知你的团队你正在进行的工作，并告诉你为了完成这个工单你需要创建的分支数量。
(2) 从 *develop* 分支上创建一个命名包含工单编号和大致的工作描述的新分支。如果你正在进行的工单拥有子任务，这个工单在你的工单系统中可能称为元工单或 Epic 工单。如果你的工作只是未来工作的一部分，你应该使用相关最小的工单号。你的工单系统可能会将这些工单称为用户故事、issue 或 bug 工单。
(3) 在你开始工作后，确保每次可能将更改并入 *master* 分支之前，工单分支能够保持最新状态。每个提交说明都以包含工单号的方括号开头：`[#1234]`。
(4) 运行相关的测试，保证你的代码中的拼写错误和常见错误都能被发现。这一步可能包括一个拼写检查、一个语法检查（通过 Lint 进行静态语法检查）。如果你在测试驱动的环境中工作，那么一定会有额外的测试需要运行。
(5) 当你完成工作（或是你认为你完成了！）的时候，上传最后一个提交，包含关键词 resolves（解决），后跟工单号：`Resolves #1234`。
(6) 可以选择将你的工单分支推送到代码托管仓库。使用提交说明中的关键字，这一推送会将你的工单跟踪工具带入下一步。
(7) 用你的工单跟踪工具为这个工单添加一条评论，说明你使用某个方法的原因，并证明工作已经完成。例如，添加一张屏幕截图，显示在你的本地开发环境中这个工单引起了哪些变化。如果无法正常工作了，这一步在后面就会变成可用性测试。
(8) 确保工单分支保持最新，然后将你的工作合并到 *develop* 分支，假设没有合并冲突，将更新后的分支推送到中央仓库。
(9) 假设新添加的工作没有带来任何新的问题（*回归*），这个工单就可以关闭了。
(10) 最后，删除你本地的工单分支和这个工单分支的远程副本。

在一些工单系统中，添加一个井号（`#`）会自动将提交说明和工单号关联起来。在工单号周围添加方括号可以保证在你选择变基你的工作时，提交说明不会被忽略，因为在自动生成新的提交对象时，以 # 开头的行会变成注释而被忽略。在很多系统中，包含关键词 resolves 将会自动把工单从正在进行移至下一状态（例如，需要测试或关闭）。这一点取决于你在使用的工单系统，不管你使用哪个系统，查看它的文档即可。

这个模式对于没有同行评审的小团队来说表现非常好。随着团队开始扩大或者需要经历特定的质量保证过程，你会发现这个模式并不能完全满足你的要求。

4.3.1 使用同行评审的可信开发者

这个工作流与最基本的团队工作流相比，增加了一个同行评审环节。现在，每个工单都由另一个团队成员从代码的角度进行评审。使用同行评审测试，而不只是自动化测试（或测试驱动开发）的原因将在第 8 章中介绍。

关键特征包括以下这些。

- 治理模型：贡献者共同维护
- 集成合并：由审查者执行
- 集成分支：*develop*

这个工作流如下所示。

(1) 当你开始一个工单时，在工单跟踪工具中更新状态，表示这个工单已经在进行中了。
(2) 从 *develop* 分支的本地副本上创建一个新的分支。
(3) 进行你的工单，使用变基来确保你的分支是最新的。每个提交说明都以包含工单号的方括号开头（`[#1234]`），或是以关键词 resolves（解决）开头，后跟工单号：`Resolves #1234`。
(4) 运行相关的测试，保证你的代码中的拼写错误和常见错误都能被发现。这一步可能包括一个拼写检查、一个语法检查（通过 Lint 进行静态语法检查）。如果你在测试驱动的环境中工作，一定会有额外的测试需要运行。
(5) 将你的代码推送到代码托管仓库。这是你的备份，所以不要跳过这一步。
(6) 当你完成这个工单时，确保工单分支与 *develop* 保持同步，然后将你的工作上传至代码托管系统。在工单跟踪工具中将工单标记为需要测试。

如果需要手动评审，而且没有完善的自动化测试，评审者将要完成以下剩余步骤。

(1) 根据原始的工单描述对工作进行一次评审。编码者有义务保证自己的工作是清晰的，与测试工作的步骤是密切相关的。如果有需要，将这个工单发回开发者进行必要的修改，或者在与 *develop* 失去同步时更新分支。
(2) 合并工单分支和 *develop* 分支，假设没有合并冲突，将更新后的分支推送到中央仓库。
(3) 假设没有出现回归，评审者会关闭工单，通知开发者他的工作已经并入主分支。开发者和评审者现在都可以删除他们本地的工单分支的副本。因为他们现在正在清理，所以评审者应该删除工单分支的远程副本，开发者可能需要停下当前的工作来完成清理。无论何时，我们都应该保持专注，维护流程，保护团队成员。

当团队大到拥有评审流程时，建立一台共享的开发服务器就很有用了，团队可以实施定期的工作展示。在开发过程中，开发服务器同时可以兼做质量保证机器。为了减少团队成员检查 *develop* 分支的最新版本以及构建软件的开销，你可以选择搭建一个 Jenkins（http://jenkins-ci.org/）实例来使这个流程自动化。

4.3.2　需要质量保证团队的不可信开发者

这个过程在 4.3.1 节之上做了一些微小的变化。这次，我们假设有一个不可信的开发者，不允许将任何人的工作并入主分支。相反，一个可信的质量保证团队执行最终的合并。

关键特征包括以下这些。

- 治理模型：具有并置仓库的贡献者
- 集成合并：由评审者执行
- 集成分支：*develop*

开发者在代码托管系统中派生一份项目来开始工作，然后在本地克隆这份仓库的派生副本。这一步只执行一次。

这个工作流如下所示。

(1) 当你开始一个工单时，在工单跟踪工具中更新状态，表示这个工单已经在进行中了。
(2) 从 *develop* 分支的本地副本上创建一个新的分支。
(3) 进行你的工单，使用变基来确保你的分支是最新的。将你的工单分支推送到你派生的项目中，作为进行中工作的备份。
(4) 运行相关的测试，保证你的代码中的拼写错误和常见错误都能被发现。这一步可能包括一个拼写检查、一个语法检查（通过 Lint 进行静态语法检查）。如果你在测试驱动的环境中工作，一定会有额外的测试需要运行。
(5) 当你完成这个工单时，确保工单分支与 *develop* 保持同步，然后将你的工作推送到你派生的仓库中。为你的工作打开一个拉取请求（在一些工单系统中，它可能称为*合并请求*）。

评审者将要完成以下剩余步骤。

(1) 根据原始的工单描述对工作进行一次评审。
(2) 在仓库的主副本中接受拉取请求。根据你的工单系统，这一步可能在网页的图形化界面上完成，也可能在仓库的本地克隆中完成。
(3) 假设没有出现回归，评审者关闭工单。因为工作是在派生项目中完成的，所以主仓库中不需要额外的清理。

如果你的团队中大多数都是可信的开发者，但也有一些合约雇员，这个方法同样表现良好。你可能会希望你的合约雇员在派生仓库中工作，而不是给他们主项目的写入权限。在一些类型的软件中，甚至你的员工之间也有这样的分割。例如，如果你在为一个医疗设备开发固件，则可能需要遵守非常严格的政府规定，限制工作雇员的权限提交，在将工作添加至仓库之前确保评审的执行。

4.4 根据计划发布软件

我工作过的大部分项目使用发布计划来将新版本软件呈现给用户。本节介绍的这个过程几乎完全基于流行的 GitFlow（http://nvie.com/posts/a-successful-git-branching-model/）工作流。如果你应用了持续集成，没有将多个工单的工作放入一次发布中，那么可以跳过本节内容。

4.4.1 发布稳定版本

至此为止，所有的例子中我们都在 *develop* 分支上工作。不过，最终你要将一直工作的产品发布出去。当你准备好发布时，需要将你的仓库分成一个面向公众的稳定版本，和一个面向开发者的“无保证”的版本。

这个工作流如下所示。

(1) 从一个基本的提交开始，创建一个名为 *master* 的新分支。
(2) 使用易于记忆的命名约定给这个提交打上一个版本号标签。例如，v1.0。
(3) 将更新后的仓库推送到集中式代码托管系统。如果没有使用自动化构建过程，就使用新的代码更新相关的服务器。

在第一个发布完成后，你现在可以将工作切分为稳定、面向公众的工作和正在进行的开发。

4.4.2 正在进行的开发

在产品的官方发布之后，你的团队将会立即开始同时考虑两个不同的环境：监控线上代码的健康状况；同时继续开发，添加新的功能或改善已有的功能。

我的偏好依然是使用短小的冲刺周期。看到自己的工作成果，开发者会充满动力。冲刺周期越长，等待别人接触到他们的工作成果的时间就越长。

我通常使用的一周发布计划有着固定的路线。对于不同团队来说，天数可能有所差别，但这个通用的方法对很多团队来说都是一个好的出发点。

设置（周一）的工作内容如下所示。

- 所有 *develop* 分支上的工作并入测试分支 *qa*。周一前没有完成及没有经过同行评审的工作只需要留在原来的工单分支即可。
- 将测试服务器更新至 *qa* 分支的最新版本。
- 为上周完成的所有用户故事创建一个 QA 清单。使用标准化的工单格式很容易编写这个清单。

你可能希望在一个单独的文档中编写你的 QA 清单，如 Google Doc 或内部 Wiki。我同时使用 JIRA 中保存的查询来寻找上周已解决的工单，或被标记到某个发布的工单。这完全取决于你在你的工单系统中如何选择跟踪流程。

测试（周一和周二）的工作内容如下所示。

- 运行自动化测试，确保关键业务的交互没有出现回归（网站访问者和会员仍然可以使用预期的功能）。
- 团队成员负责测试清单中的所有项目,并将工单的状态更新为PASS(通过)或FAIL(失败)。
- 任何发现的 bug 都应该打开新的工单，并在发布日前提出，要么通过新的修复解决，要么从 *qa* 分支上移除相关的提交。

发布日（周三）的工作内容如下所示。

- 将 *qa* 分支并入 *master* 分支，打上发布版本的标签。
- 通过签出 *master* 分支上的提交，也就是项目最新的发布，更新线上网站。使用显式标签确保你可以轻松退回到先前的状态。

将新功能和修复的公告发布到开发博客。很多团队选择在发布日后等待一至两天再发布博客上的文章。团队能够确定这个发布稳定，不需要回滚。

4.4.3 发布后的补丁

有时部署的代码存在错误。如果一个 bug 需要在下次软件发布前得到快速修复，这时就需要进行一个计划外的修复。这些部署通常称为补丁。

在补丁中，工作不是从 *develop* 分支开始，而是从 *master* 分支开始，确保更改只引入了部署代码中发现的问题的修复。

这个工作流如下所示。

(1) 创建一个补丁分支，签出 *master* 分支，而不是 *develop* 分支，确保没有意外地将额外的功能加入这次修复。
(2) 生成标签名的列表，定位到最新打上标签的发布。
(3) 从 *master* 最新打上标签的发布上，创建一个新的分支，使用 `hotfix-` *<ticket_number>*-*<description>* 格式的分支名。例如，`hotfix-1234-fixing_three_seat_issue`。
(4) 完成与开发工单相同的评审步骤。
(5) 将通过测试的补丁分支并入 *master* 分支。
(6) 将 *master* 分支上的最新提交打上最新发布的版本号。例如，v1.0.1。
(7) 将经过测试的补丁分支并入开发分支，保证在下次软件正式发布时这些修改不会丢失。

4.5 非软件项目中的协作

Git 不只是软件开发者的专利！作为技术作家，我使用 Git 来跟踪不是软件的文件，例如，配置文件、正在编写的文章，甚至是这本书！有人甚至使用 Git 来维护个人日记。为了说明将 Git 命令与团队流程一一对应的重要性，让我来解释一下我是如何组织这本书的仓库的。

在写这本书时，我使用的是 O'Reilly 的自动化构建工具 Atlas（https://atlas.oreilly.com/）。这个系统还提供了一个网页，使得编辑可以直接在上面操作图书文件，保存的文件被立即提交到 *master* 分支。界面上没有同行评审流程，因为我的团队中任何人都可以直接编辑一份文件。然而，我偏好在本地工作，而不是通过网页。最初，我一直在 *master* 上工作，本地的分支开销很低。直到一次出现的合并冲突改变了我在本地的工作方式。

当我想要更新我的工作时，我会使用 `fetch` 命令来查看我的编辑做出的修改。通过 `fetch` 命令，我将我的 *master* 分支的副本与他们的 *master* 分支的副本（*origin/master*）相比较。假设我同意编辑做出的所有修改，我会把他们的分支副本并入我本地的副本；如果我有异议，我会使用 `--strategy-option=ours` 策略来合并他们的分支，即丢掉他们的修改，但是让 Git 以为这两个分支已经合并。

上面的做法是基于提交层面的，如果出现了合并冲突，你还可以使用合并工具来进行细粒度的逐行修改。（将东西直接丢掉听上去有些像被动的攻击，但这只是单分支系统的限制，你无法在单独的分支中讨论提议的更改。）根据提交的粒度，我或许还会选择 `cherry-pick` 一些命令（并保留他们），但丢弃其他提交。

然后你开始收到 PDF 格式的评审意见，我又一次意识到，我还有另一种分离工作的方式。我想要新写一章，并保持这些提交干净整洁，但有时候我一章正写到一半，进来了一份编辑修改，我希望及时处理。我设置了如下的分支结构：*master*、*drafts*，以及每章单独的一个分支。

drafts 分支让我能够整合我正在进行的所有工作。它通过合并已完成的章节来保持同步，或是在编辑进行修改后变基 *master* 分支。当我自己编写章节，没有其他人的贡献时，多分

支会产生高昂的维护成本，但当越来越多的贡献者开始提供不同类型的贡献时，更细的分支粒度使我能够选择如何更新我的稿子。

正如你所看到的，我的流程和软件项目使用的工作流截然不同，但我用的同样是 Git！你的工作可能有自己的特点，需要非标准化的分支。不要害怕尝试，当你尝试时，将你的流程记录下来，让别人能够理解预期的影响。

4.6 小结

本章描述的工作流在我工作过的团队中已经取得了成功的实践。你自己的团队或许希望进行一些调整，但有这样一个参考总比从零开始要好。

- 你使用的工作流在产品发布前后可能变动。
- 在发布前，你可能拥有更少的集成分支，因为不太可能出现补丁的概念。
- 通过使用你的文档来完成工作，你确保你的文档总是最新的，这样在你需要帮助时，其他同事可以更快上手。

在第二部分中，你将学到实现本章描述的流程需要的命令。

邮
电

第二部分

在工作流中使用命令

在本书的这一部分中，我们从实践的角度来学习 Git 命令。首先会给出一个场景，然后会给你一些让自己陷入（摆脱）困境的命令。

本章随处可见需要上手实践的活动。在可能的情况下，你应该完成这些活动，因为这能帮助你深入理解命令（并且在从事自己的软件项目时，你也会觉得相应的消息更加自然）。当提供了图表时，你应该自己重新绘制一遍，因为每当你学习一个新的活动时，你的一举一动都会帮助你在脑海中强化这些信息。

在阅读后续章节前，你应该确保你安装了最新版本的 Git（参见附录 B），并确保你的系统已经配置正确（参见附录 C）。

第 5 章

单人团队

尽管这本书是面向多人团队的，但我们经常也会单兵奋战，即作为独立开发者。可能你是在做自己的业务项目，或者你的团队中只有你一个开发者。没有团队约束的单兵奋战或许有些吓人，因为没有人会告诉你应该做什么，或是在你陷入困境时帮你一把。我会向你展示我在做自己的项目时是如何工作的。当然，有些时候我偷懒省去了一些步骤。（毕竟没有人监督我，所以谁会知道我走了捷径呢？）在可能的地方，我将会向你展示这些捷径的效果。

学完本章之后，你将具备以下技能。

- 创建一份远程仓库的本地副本
- 为现有的文件集初始化版本控制
- 从空的项目目录开始创建一个新的仓库
- 通过提交消息来检查仓库历史
- 通过分支来隔离无关的工作
- 在本地仓库中创建提交
- 使用标签来高亮单个提交
- 建立你的项目与远程代码托管系统的连接

如果你是一个创造者（相对于评审者或管理者），那么你的大部分时间应该花在使用本章介绍的命令上。应当将熟练使用这里介绍的所有工具视为阅读本部分剩余章节的前提。

喜欢通过视频教程学习的读者可以参考本书附带的系列视频，Collaborating with Git（http://shop.oreilly.com/product/0636920034872.do）。

5.1 基于issue的版本控制

曾经有人和我说，最擅长描述一个问题的人最有可能能够解决它。在写这本书的时候，我发觉这个观点太正确了。当我给自己写下一个模糊不清的待办事项（TODO），比如“完成第 4 章”时，我感到几乎没有继续写下去的动力。但当我把这个任务描述为“为 Mai 这样的小型团队编写样例工作流”后，我更有深入写作的动力。不过，这不仅限于写书。作为单人团队，你可能没有很大的动力来编写代码。不过，要是你和我一样，如果做的事情能够帮助别人，你就会更有动力把这件事情做成。

> 如果你从未思考过是什么激励你成为一个开发者，你或许会喜欢 Joe Shindelar 的演讲“开发者眼中的开发者管理”（https://austin2014.drupal.org/session/developers-primer-managing-developers.html）。

你或许会问自己“这个和 Git 有什么关系”。每次你坐下来，在使用源码管理的项目中工作时，你应该对你所做的事情了如指掌。无论你是在开发一个新功能，修复一个 bug，重构旧的代码，还是在尝试新的想法，你都应该有某种改进动力。你有很多种方式将想做的事情写下来，但下面介绍的方式更加优雅，与只是用到工单相比，回报也更丰厚。

工单包含以下三个主要部分。

- *问题*
  大致描述你想要解决的问题。
- *原因*
  为什么你想要做这件事（如果问题得到解决，谁将受益）？
- *质量保证测试*
  我将如何知道这个问题已经得到解决？

这种格式与我所见到的敏捷项目使用的另一种格式非常相似。

- *卡片*
  从用户的角度，大致描述这个问题。
- *对话*
  你想要解决的问题的细节。在可能的情况下，应该避免给出具体的解决方案。
- *确认*
  （第一部分中的）用户用于验证问题是否解决的步骤。

在单人团队中，你可能感觉使用工单的开销对你来说大了些。或许你的记事本就已经足够了。我通常一开始也这样认为，但继续做项目时，我开始不断地忘记一些微小的想法。有时我为每个想法创建一个新的分支，但最后被一大堆过时的分支压得透不过气来。如果你有同感，请现在就花点时间，找到你使用的代码托管系统中的工单跟踪选项，然后渐渐养成习惯，记录你将对你的软件所做的工作。至少，它会给你一些用于创建分支和跟踪代码的编号。

如果你目前还没有代码托管系统，我向你推荐 GitLab，或者它的免费在线版本：GitLab.com。你可以免费创建不限合作者数量的私有仓库，如果你在防火墙内学习 Git，还可以将它安装在局域网中。私有仓库的优点是你可以在学习时隐藏你的工作。如果别人看不到你的工作，你将不能得到社区的帮助，但我觉得你可能现在还有些害羞，不会去社区里交流。对于我们大多数人来说都是这样的。

一旦你找到一种跟踪想法的方式，工作的流程应该遵循以下步骤。

(1) 在你的 issue 跟踪系统中创建一个新的工单，注明这个 issue 的编号。
(2) 在你的本地仓库中，使用 *issuenumber-description* 格式创建一个新的分支。
(3) 完成工单描述的工作（且只完成工单中描述的工作）。
(4) 测试你的工作，确保已经完成并且是正确的。确保它能够通过开发环境下的 QA 测试。
(5) 现在，你有了一个“杂乱”的工作目录，其中包含了新增的文件和（或）修改过的文件。将你的更改添加到本地仓库的暂存区。
(6) 将你缓存的修改提交至仓库。
(7) 将你的更改推送到备用服务器上。在很多情况下，这也将是追踪你的工单的地方，如 GitLab、Bitbucket 或 GitHub。根据你的工单系统，现在可以将这个工单标记为已解决，但不需要将其标记为已关闭。
(8) 当你对你的工作完全满意时，将你的工单分支并入主分支（通常是 *master*）并将修改后的分支推送到代码托管系统中。
(9) 再一次测试你的工作，确保没有后续问题。
(10) 将你的工单标记为已关闭。

根据你正在编写的代码类型，这些步骤可能略有不同。推翻重写这个列表，加入与你工作方式不同的所有步骤。例如，你可能使用测试驱动开发的实践，或是使用构建脚本来部署代码。使用你自己的流程。如果你不习惯使用文字，请画出你的流程（图 5-1）。

不管你选择怎么做，确保记下你的流程。你可以选择在仓库的 README 文件中记录这些，或是打印出来并贴在你的看板上。通过遵守一致性约定，与同事一起建立每个人都能遵守的流程将变得无比轻松。

在本章剩余部分中，你将学到之前描述过的流程中需要用到的命令。我们将会从创建新的仓库开始，你可以在此存放你的工作。

5.2　创建本地仓库

当你在 Git 中创建一个新的仓库时，一般会从以下三个起点之一开始创建。

- 一个克隆的仓库
- 一个文件未被跟踪的文件夹
- 一个空目录

在本节中，你将会学到如何使用这三种方法创建一个新的仓库。

开始时，创建一个文件夹，用于存储你所有的样例仓库（例 5-1）。你可以选择将这个文件

夹放在你的桌面上、你的 home 目录中，或是其他什么地方。Git 不关心这个文件夹的位置，只要你自己记得这个文件夹的位置即可。

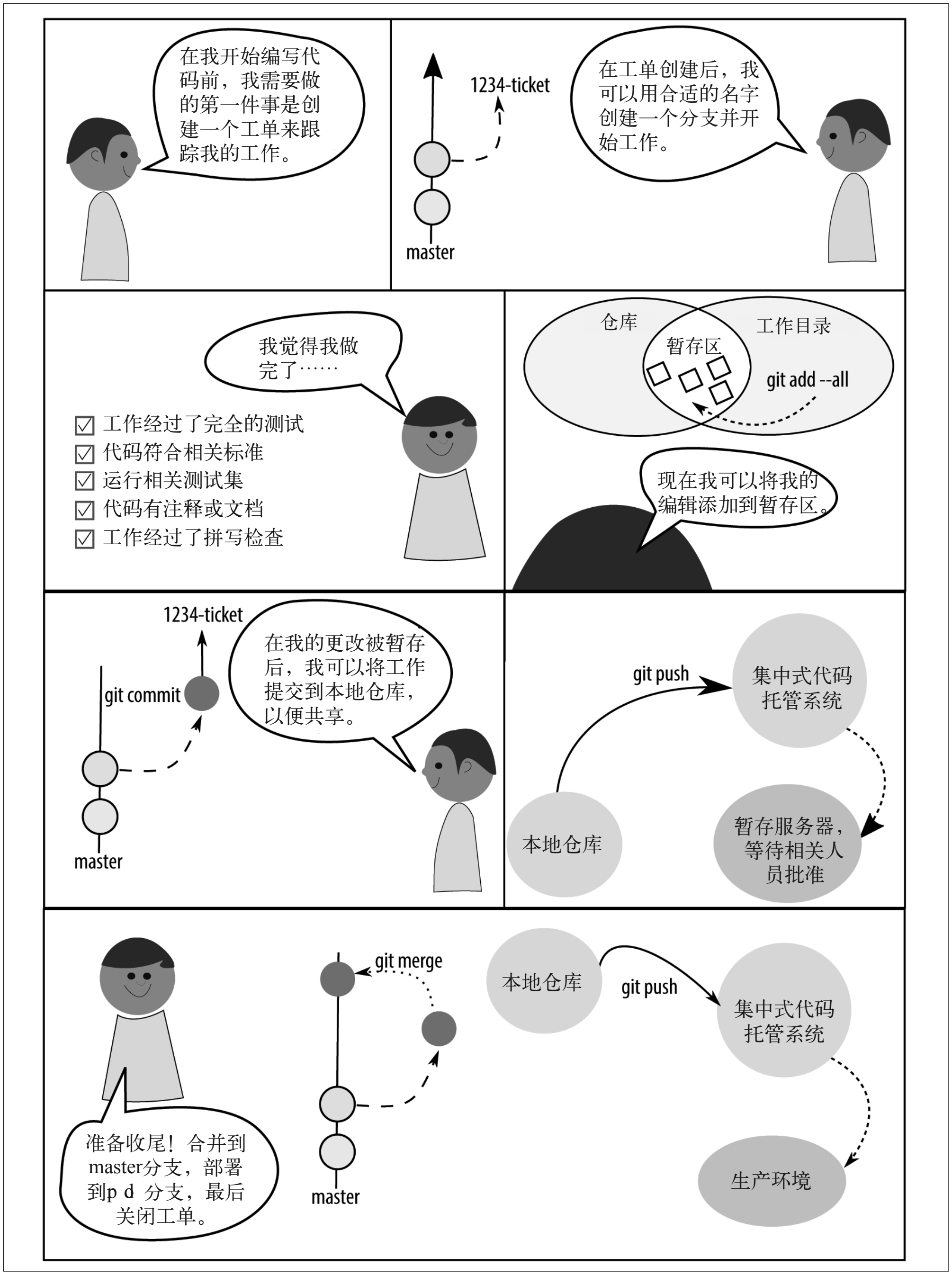


图 5-1：勾勒你的工作流图

例 5-1　在 home 目录下创建一个项目目录

```
$ mkdir learning-git-for-teams
$ cd learning-git-for-teams
```

除非特别说明，本书所有练习都会假设你当前正在访问该文件夹中的一个样例仓库。如果需要使用特定的仓库，我将会在说明中指出。但一般来说，使用哪个仓库并不重要。

5.2.1　克隆已有的项目

在代码托管系统（例如 GitLab 或 GitHub）上，当你访问一个项目页面时，通常能够找到一个选项来将所有文件下载为 .zip 压缩包，或者创建一个仓库的克隆。这些选项通常都在一起，但并非一定在一起。图 5-2 显示了 GitLab 中仓库 URL 的地址。

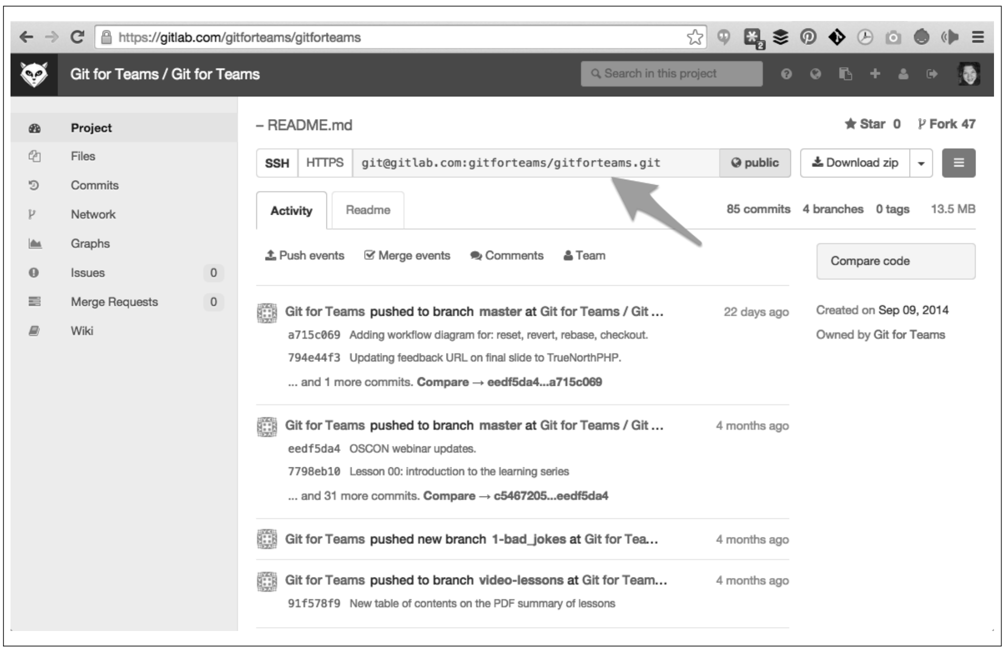


图 5-2：克隆仓库使用的 URL

> **熟能生巧**
>
> 如果使用一个已有的仓库来学习 Git，那么你学习命令时将会更轻松，不会制造出一大堆需要解决的问题。

为了下载一个项目的副本，你将会使用 `clone` 命令，如例 5-2 显示。与下载压缩包不同，创建项目克隆将会下载仓库中所有文件的副本及其提交历史，它还会记住你下载代码的地方，将远程代码托管服务器设置为跟踪仓库。不要担心，它不会永久保持连接，它只是收藏了这个位置，以备你以后想要检查更新，并下载到你的本地仓库。

一个项目只需要克隆一次。一旦项目下载完成后，你将会使用一系列不同的命令来使它保持同步。在第 7 章中，你将会学到 `clone` 命令的不同使用方式。在本章中，我们只需要用它来抓取项目的一份快照，以使你可以在某个起点上开始工作。

例 5-2　创建“Git 团队协作”仓库的一个克隆

```
$ git clone https://gitlab.com/gitforteams/gitforteams.git
```

接下来，你的命令行窗口中将会出现下面的输出。

```
Cloning into 'gitforteams'...
remote: Counting objects: 1040, done.
remote: Compressing objects: 100% (449/449), done.
remote: Total 1040 (delta 603), reused 915 (delta 538)
Receiving objects: 100% (1040/1040), 9.49 MiB | 1.68 MiB/s, done.
Resolving deltas: 100% (603/603), done.
Checking connectivity... done.
```

恭喜你！你已经成功克隆了你的第一个 Git 仓库。你可以在这个目录中随意折腾。如果你把这个目录弄得面目全非，删除文件夹并重新运行 `clone` 命令即可。

现在，在有了这个目录之后，你同样拥有了本书需要的所有帮助资料。你可以浏览这些帮助文件，寻找隐藏的彩蛋，随意找些什么东西开始看，这样在你学到更多高级命令时不必担心会制造出奇怪的问题，或是摧毁你自己的工作。

5.2.2　将已有的项目迁移至Git

如果我第一次参与软件项目，我倾向于将文件打包下载，然后在软件第一次导入的某个时间点开始做版本管理。我将会简明扼要地推进工作，做一个棘手的自我介绍，讲述我要怎样保持开发者最初想要的样子。

为了比较你正在运行的命令的作用，请再下载“Git 团队协作”仓库，不过这次是从你刚克隆过的那个仓库中下载一个压缩包，步骤如下所示。

(1) 访问 https://gitlab.com/gitforteams/gitforteams。
(2) 找到并下载项目的压缩包。
(3) 解压项目，放到你为本书准备的项目目录下。因为你在这个目录中已经有了一份克隆的文件副本，所以你应该将这个新建的文件夹命名为 gitforteams-zip。

你可以在任何文件夹下使用初始化命令 `init` 创建一个 Git 仓库，如例 5-3 所示。Git 将会感知到这个目录中的所有文件，包括子目录，从而确保你是在项目的根目录下运行 `init` 命令。

例 5-3　初始化目录的版本控制

```
$ git init
```

你将会看到类似以下内容的一条消息。

```
Initialized empty Git repository in /Users/emmajane/gitforteams/gitforteams-zip/.git/
```

系统并没有立即将文件加入仓库。这是 Git 的一个特性，因为它同样允许你忽略文件，你需要主动告诉它你想要跟踪哪些文件。如果存在相关的下一个步骤，Git 几乎总是会在状

态（status）消息中显示有用的建议。你应该养成经常使用 `status` 命令的习惯，就像在文字处理程序中使用保存那么频繁。这个命令不会保存你的工作，但它让你知道当前仓库中发生了什么。知道仓库中发生了什么是理解 Git 的关键。快去查看你的仓库的当前状态（例 5-4）。

例 5-4　检查仓库的当前状态

```
$ git status
```

Git 告诉你，下一步是添加你想要跟踪的文件，因为你刚刚初始化了仓库，如下所示。

```
On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        [ lots of files listed here ... ]

nothing added to commit but untracked files present (use "git add" to track)
```

将文件导入 Git 分为两个步骤。尽管你最初会觉得有些厌烦，这个功能允许你在工作目录中同时进行多个无关的更改。更改可以被*暂存*在索引（index）的一组提交中，每组都有一个不同的提交消息。我们希望添加工作目录中的所有文件，因为这是我们第一次导入文件（例 5-5）。

例 5-5　将仓库中所有文件添加至暂存区

```
$ git add --all
```

同样，可以使用 `status` 命令来检查当前状态。这个命令的输出将会提示你这些文件已经被暂存并已准备好提交，如下所示。

```
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   [ lots of files listed ... ]
```

现在，你的文件已经添加完毕，你可以使用 `commit` 命令将你当前的状态保存到仓库（例 5-6）。

例 5-6　将所有暂存的文件提交至仓库

```
$ git commit -m "Initial import of all project files."
```

此时会在屏幕上输出一个较长的提交确认信息，提醒你已经将这些文件加入你的仓库。你的项目文件现在已经处于版本控制下。

5.2.3　初始化空项目

在我们教学 Git 时，从一个什么都没有的空目录开始是最容易的，因为讲师和学生很容易从一个相同的起点开始。下面这个练习能让你轻松地初识 Git。

(1) 创建一个新的空文件夹，如下所示。

```
$ mkdir empty-repository
```

(2) 进入新创建的文件夹，如下所示。

```
$ cd empty-repository
```

(3) 运行 Git 初始化命令，如下所示。

```
$ git init
```

(4) 确认已添加隐藏的仓库文件夹，如下所示。

```
$ ls -al
```

在 Windows 上的命令如下所示。

```
dir
```

如果你看到了一个新的隐藏文件夹 .git，那么你的仓库已经成功创建。这个文件夹将会包含你仓库中所有更改的记录。这个文件夹里没有什么可怕的东西，但如果你删除了它，那么系统将无法继续跟踪你的项目。也就是说，你将不再能够在你的仓库中恢复任何文件之前的版本，并将丢失仓库中所有的提交消息，而文件当前的状态也将不可变。

现在，你可以遵循上节中的额外步骤来添加文件（例 5-5），并将它们提交至你的仓库（例 5-6）。

5.2.4　查看历史记录

一旦你完成了仓库中的第一个提交，就可以开始查看历史记录了。当然，你的项目历史是你已经完成的工作和其他协作者完成的工作的组合。如果你从未下载过开源项目，那么几乎感受不到协作的存在，但它确实存在。协作可以简单到只是在别人的工作上添加你的更改。

要查看仓库中产生的更改，请使用 `log` 命令（例 5-7）。默认情况下，这个命令允许你查看本地仓库当前签出分支中的每个提交的提交消息和作者信息。

例 5-7　使用 `log` 命令查看仓库历史记录

```
$ git log
```

`log` 命令将会按时间倒序输出仓库中提交消息的完整历史记录。

确保你的详细信息已经进行配置

如果你的名字和邮件地址没有显示出来，参见附录 C 来了解如何配置 Git。

如果你只产生了一条提交消息，也就是最初的一次导入，那么只有一条消息会被显示出来，如下所示。

```
commit fa04c309e3bb8de33f77c54c1f6cc46dc520c2ca
Author: emmajane <emma@emmajane.net>
Date:   Sat Oct 25 12:44:39 2014 +0100

    Initial import of all project files.
```

但如果你在一个更完善的代码库上工作，那么代码库里面将会有很多消息。看完这些消息所需的工作量将会是巨大且复杂的。你可以通过添加 `--oneline` 参数来缩短消息，只显示消息的第一行，如例 5-8 所示。按 `q` 退出。

例 5-8　查看缩短的项目历史记录

```
$ git log --oneline
```

为了理解同一个文件如何拥有不同的历史记录，在克隆的仓库和通过下载的 .zip 包创建的仓库中，分别运行例 5-7 和例 5-8 中的命令。即使这些文件是相同的，它们的历史记录也并不相同（在第 6 章讲到变基时会再次提到）。

> 其他分支会有不同的提交，而且不同的仓库副本中会包含不同开发者产生的提交。事实上这很混乱，但混乱仅存在于每个仓库内。作为软件团队，我们创建的约定可帮助我们在混乱中重建秩序，并允许我们以相同的行为共享工作。（记得第 3 章中提到的分支策略吗？它们将会保持工作按照符合逻辑的线性排序。记得第 2 章中的权限策略吗？它们将会把人们锁定在正确的位置上，并且无法在没有社区看门人同意的情况下对“幸运”仓库进行更改。）

如果你完成了本节中的所有步骤，现在就会有三个独立的仓库来完成剩余的活动。对于有关分支的部分，我建议你使用克隆的仓库，因为它有更多分支可供你查看。对于其他部分，你可以选择三者中的任意一种。

5.3　使用分支工作

在版本控制中，分支是分离不同想法的方式。分支有很多种用法。你可以使用分支来标记软件的不同版本。你可能会使用临时的分支来解决一个 bug，或者使用长期运行的分支来测试一个新的想法。

5.3.1　列出分支

要列出所有分支（例 5-9），你可以使用不加参数的 `branch` 命令，或者添加 `--list` 参数。在本章开始时，你克隆过一个仓库；在本节中将使用这个仓库，因为它已经包含了你需要查看的分支。

例 5-9　列出本地分支

```
$ git branch --list
```

本地分支将以列表的形式输出，如下所示。

```
master
```

在默认情况下，会将 *master* 分支复制到你的本地仓库中，然后你可以直接在这个分支上工作。除了这个分支以外，你还下载了远程仓库中所有其他的已有分支。你可以引用这些分支，但在设置好远程分支的工作副本之前，将无法在这些分支上工作。要列出仓库中的所有分支，请使用 `--all` 参数（例 5-10）。

例 5-10　列出所有分支

```
$ git branch --all
```

如果你在克隆仓库的本地副本中执行这个命令，应该会同时看到你的本地分支和一个远程分支。* 表示你当前正在查看（或“签出”）的分支。剩下几行以 `remotes/`*origin* 开头：`remotes` 仅表示“不在本地”，而 *origin* 是一种默认约定，表示“我的副本是从这克隆的”。最后一部分是分支名（`master`、`sandbox` 和 `video-lessons` 就是所有的分支了），如下所示。

```
* master
  remotes/origin/master
  remotes/origin/sandbox
  remotes/origin/video-lessons
```

不过，这个列表可能会有些误导。事实上，远程分支的名称中不包含 `remotes` 这个词。它只是用来告诉你它所对应的分支类型。要获得可用的远程分支名的列表，请使用 `--remotes` 参数（或缩略为 `-r`，例 5-11）。

例 5-11　列出远程分支

```
$ git branch --remotes
```

这个命令将会给你一个仅包含远程分支的列表（使用这些分支的真实名称），如下所示。

```
origin/master
origin/sandbox
origin/video-lessons
```

你可以访问到这些分支，尽管你需要在向它们提交更改之前创建你自己的副本。

5.3.2　更新远程分支列表

远程分支列表不会自动更新，因此这个列表将会随着时间而落后。使用 `fetch` 命令更新这个列表（例 5-12）。

例 5-12　获取更新的列表和所有远程分支的内容

```
$ git fetch
```

你将在第 7 章中学到更多与远程工作相关的内容。

5.3.3　使用不同的分支

当你签出一个分支时，你更新了系统（工作区）中的可见文件，来匹配仓库中存储的版

本。这个切换是通过 checkout 命令完成的（例 5-13）。签出过程与 SVN 等集中式版本管理系统（VCS）有些不同。在集中式 VCS 中，由于分支并没有存储在本地，你需要网络连接才能使用 checkout 命令，而且必须在使用之前下载这些分支。

例 5-13　使用 checkout 命令切换分支

```
$ git checkout --track origin/video-lessons

Branch video-lessons set up to track remote branch video-lessons from origin.
Switched to a new branch 'video-lessons'
```

在旧版本的 Git 中，这个命令的工作方式有些不同。如果前置命令报错，你可能会选择升级（参见附录 B），或运行下面的变种命令。

```
$ git checkout --track -b video-lessons origin/video-lessons
```

这个命令（checkout -b）启用了跟踪（--track），从远程仓库 origin 中存储的 video-lessons 分支上，创建了一个名为 video-lessons 的新分支。这个远程分支的本地副本可以通过 origin/video-lessons 访问到，而你自己的分支副本可以通过 video-lessons 访问到。

你现在应该已经有了远程分支 video-lesson 的一份本地副本（图 5-3）。

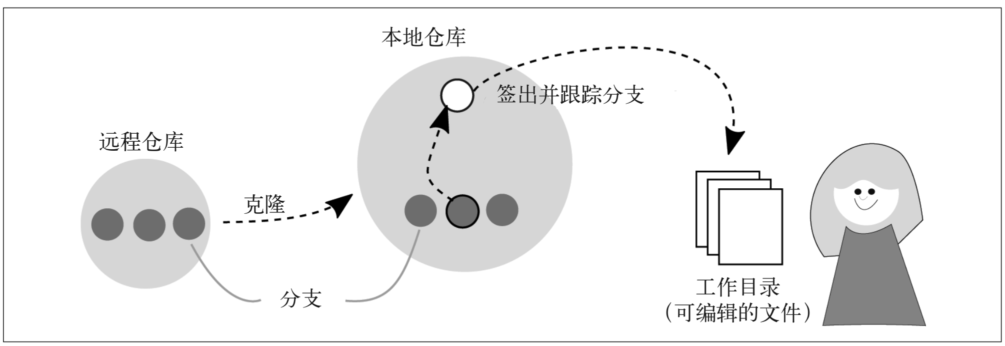


图 5-3：远程分支的本地副本已经创建

在你的分支列表中，似乎这个分支出现了两次，这是因为其中一次代表远程仓库中的引用信息，如下所示。

```
$ git branch -a

  master
* video-lessons
  remotes/origin/master
  remotes/origin/sandbox
  remotes/origin/video-lessons
```

从新的分支上，你可以使用例 5-22 或例 5-23 来查看历史记录。注意，两个分支中的提交历史是不一样的。

5.3.4　创建新的分支

对于小型项目来说，我愉快地将 *master* 分支中的每个提交都视为一个问题解决方案。但

是，当团队规模增长时，确立团队结构的协作方式将会使你越来越多地受益。第 3 章讲到了你可能想在你的团队中采用的分支策略。作为独立开发者，你或许更难明白什么时候你应该在一个不同的分支上工作。为了帮助你作出决定，请问自己以下几个问题。

- 如果进行得不顺利，我是否会想要完全丢弃这个想法？
- 我正在创造的东西是否严重偏离了当前发布的软件版本？
- 在进行发布或被软件的发布版本接受之前，我的工作是否需要经过评审？
- 完成这项工作之前，我是否有可能切换到其他任务上去？

如果你对每个问题都持有肯定的答复，你应该考虑为你的工作创建一个新的分支。现在开始培养良好的使用习惯就像是买保险一样，你希望你永远都不会用上，但总得提前做好准备。

决定什么工作需要进入新的分支的最佳方式是使用 issue 跟踪工具。通过书面描述你即将进行的工作，你将会对什么时候开始并结束你的分支看得一清二楚。是的，这通常看上去很繁琐，但这是一个值得养成的习惯，尤其是当你在更大的团队中工作时。

当你创建了一个新的分支，此时这个分支包含了与原分支相同的历史记录（图 5-4）。当你使用 `log` 命令查看新的分支的历史记录时，祖先分支中的提交也会显示出来。

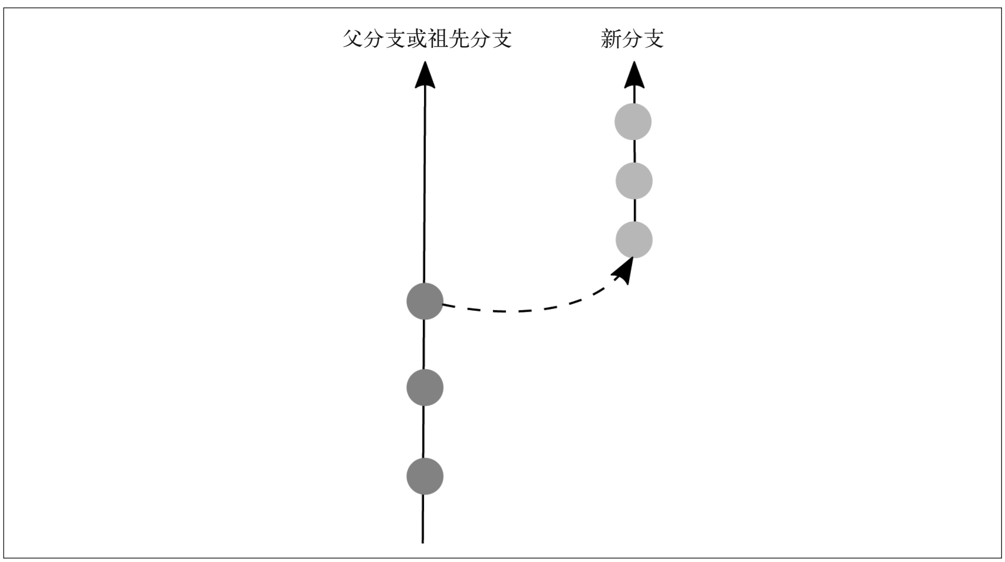


图 5-4：新的分支包含祖先分支中的相同提交

假设你在使用基于 issue 的版本控制，你的分支名应该能够反映你正在工作的工单。例如，如果 issue 是“1: Add process notes to README”（在 README 中添加过程备忘），那么分支应该被命名为 *`1-process_notes`*。新分支的历史将会包括当前所有提交，因此确保你从正确的起点开始新的分支。你可以通过 `checkout` 命令来移动到正确的分支（例 5-14），或者你可以将你想要的父分支添加到你的命令中（例 5-15）。

例 5-14　创建一个新的开发分支

首先签出你希望作为起点使用的分支，如下所示。

```
$ git checkout master

Switched to branch 'master'
```

然后，创建一个新的分支，如下所示。

```
$ git branch 1-process_notes
[没有消息显示]
```

最后，签出新分支，如下所示。

```
$ git checkout 1-process_notes

Switched to branch '1-process_notes'
```

尽管要记的东西有点多，但例 5-15 的优点是可以直接从正确的基线分支创建一个新的分支，这意味着你不需要记住之前的指令来执行额外的签出步骤。

例 5-15　从主分支创建一个新的开发分支

```
$ git checkout -b 1-process_notes master

Switched to a new branch '1-process_notes'
```

一旦你签出到新的分支，就可以继续完成你的工作。作为一个小练习，我鼓励你试着写下一些笔记，记录你在本章创建的三个仓库的一个仓库中采用了怎样的流程。在你的修改全部结束后，是时候将你的更改提交至本地仓库了。

5.4　在仓库中添加更改

每次你在工作目录中进行修改时，都需要显式地将这些更改保存到你的 Git 仓库。这个过程分为两步。图 5-5 显示应该如何将更改显式地暂存到索引，然后保存至你的仓库。

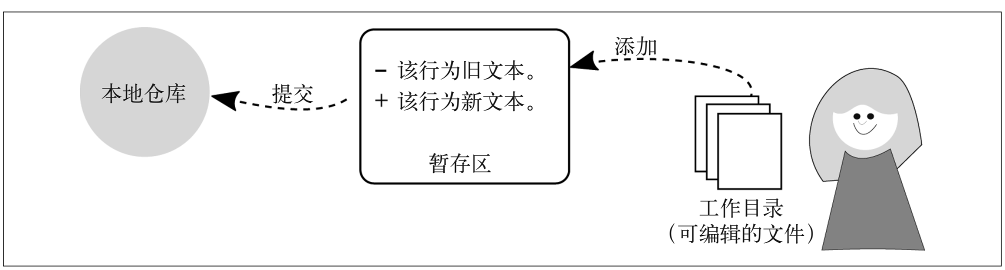


图 5-5：Git 中的更改必须先进行暂存，然后再保存至仓库

之前你创建一个新的仓库时，一次性导入了一系列文件（例 5-5）。不过，你也可以选择不这么做。当你同时进行一些不相关的编辑，并且想将这些修改放置在不同的提交中时，这个功能将会尤为有用。如果你确实希望将这些更改分离到多个提交，需要将先前使用的 `--all` 参数替换成你想要暂存的文件名（例 5-16）。你可以同时添加一个或多个文件名，文

件名不需要是同一种类型的。

例 5-16　将选中的已更改文件添加至你的 Git 仓库

```
$ git add README.md process-diagram.png

$ git add branch-naming-rules.png
```

大多数时候，我都会一次一个地将文件添加到暂存区。我发现这个习惯能够避免我意外添加更多文件。在命令行中，我可以输入文件名的前几个字母，按下 Tab 键，然后剩下的文件名会自动显示出来（这就是 Tab 补全，我的最爱之一）。但是，如果你有很多文件需要添加，而且它们还不在同一个目录下，你或许会想要使用通配符来匹配子目录中的文件（例 5-17），或者匹配文件名相似的文件（例 5-18）。

例 5-17　递归地添加指定路径中的所有文件

```
$ git add <directory_name>/*
```

例 5-18　添加扩展名为 .svg 的所有文件

```
$ git add *.svg
```

你还可以完全忽略文件名，根据 Git 中是否已知这些文件来暂存它们。通过使用 `--update` 参数，你可以暂存 Git 中所有已知的且在上次提交之后编辑过（或修改过）的文件，如下所示。

```
$ git add --update
```

如果你想要更加粗放一些，可以添加 `--all` 参数来暂存工作目录中所有修改过的文件。这个参数将会重新暂存在首次暂存后发生修改的任何文件（确保所有新的修改都包含在这个提交中。暂存 Git 中所有已知但还没有进行暂存的文件；暂存任何当前未被 Git 跟踪的文件。这是一个非常贪婪的命令！在使用前，你应该检查即将添加的文件的列表，如下所示。

```
$ git status
$ git add --all
```

一旦将变更添加至暂存区，就必须提交这个更改。如果你继续编辑任何一个刚刚添加到索引中的文件，那么在下次运行 `commit` 命令时，仅会添加之前暂存过的更改（图 5-6）。如果你继续修改某个文件，并且希望新的变更包含在提交中，就需要重复之前的命令，再次将文件添加至暂存区。

你可以运行 `commit` 命令来将暂存的变更提交至仓库（例 5-6）。

如果你在最初感到有些挫败，请记住不是只有你一个人是这样的！我花了很长时间来习惯它的行为，当它没有意识到我已经修改了文件并暂存了新的修改时，我还以为是 Git 坏了。直到开始学到部分暂存文件，我才意识到“不自动暂存我的修改”这个功能有多么强大。

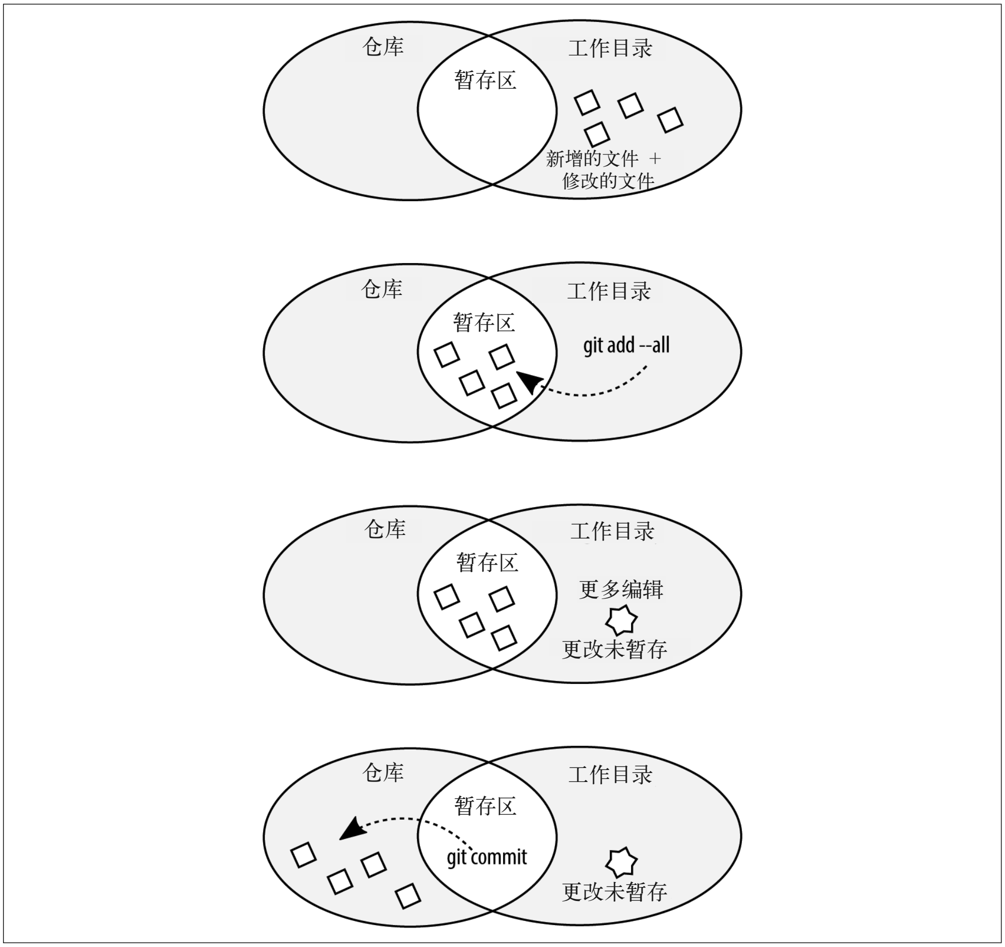


图 5-6：一次提交只会保存已添加至索引中的文件

5.4.1 在仓库中添加部分文件修改

如果你希望你的提交能有更细的粒度，那么可以选择使用 `--patch` 参数来添加已保存的文件中的部分修改。我喜欢使用这种方式提交文件，其中一个原因是，我可以将无关的编辑记录在多个更小的提交中。

通过 `--patch` 添加文件的过程分成多个步骤（例 5-19）。你将首先初始化这个过程，然后从列表中选出一个选项，来决定如何创建你的补丁。你将会看到一条提示，将这些修改添加至暂存区（`y`），或不改变这个补丁片段（hunk）（`n`）。修改过的行将会以一个 `-`（删除的行）或一个 `+`（新增的行）开头。如果某行进行过修改，它会同时显示为被删除和被新增。

为了将这些补丁片段拆分成更小的单元，你可以使用 `s` 选项来拆分这个补丁片段。只有在两个补丁片段之间存在至少一行未修改的工作时，这个选项才会生效。如果你希望将相邻

的两行分别暂存，可以编辑（e）这个补丁片段。

例 5-19　将选中的修改交互式地添加到你的 Git 仓库

```
$ git add --patch filename
```

通过添加可选的文件名，你不再需要将所有文件都看一遍。如果你确切知道想要分割哪些文件，并且有很多文件需要暂存，这个选项可以节省你大量处理单个文件的时间。在运行这个命令后，你将会开始遍历这些文件，寻找需要暂存的修改，如下所示。

```
diff --git a/ch05.asciidoc b/ch05.asciidoc
index 8f82732..e7be9ce 100644
--- a/ch05.asciidoc
+++ b/ch05.asciidoc
@@ -6,7 +6,6 @@ changed significantly in the last few years; however, a few of
the commands we'l easier to remember. Chances are very good that you have Git
installed if you are using Linux or OSX. If you are using Windows, however,
the changes are very good that Git is not installed unless you've explicitly
installed it already.

-. Open a terminal window.
 . Enter the command: +git --version+

 The version of Git you are running should be printed to the screen.

Stage this hunk [y,n,q,a,d,/,j,J,g,e,?]?
```

在这里显示的输出中，我们可以看到 Git 询问我们是否想要暂存这一行修改（`. Open a terminal window.`），也就是通过 - 指示的这一行删除。你可以按下 ? 查看处理这个补丁片段的其他选项。

5.4.2　提交部分更改

假设你刚才只将某个文件中的部分修改添加到了暂存区，那么在查看仓库状态的时候，将会看到这个文件准备好了被提交的修改，同时也包含未被暂存的修改，如下所示。

```
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   ch05.asciidoc

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   ch05.asciidoc
```

如果你将一个文件添加至暂存区，然后在提交至仓库之前继续编辑这份文件；或者如果你只选择暂存一些补丁片段，同时使用 `patch` 参数交互地将文件添加到索引，都会出现与此相同的信息。

5.4.3 从暂存区移除文件

如果你不小心向暂存区添加了太多的文件，同时想把这些修改分割为更小的提交，那么可以取消暂存你提出的修改（例 5-20）。从暂存区移除文件并不意味着撤销你所做的修改；它会通知 Git，你现在还没有准备好将这些更改提交到仓库。

例 5-20　从暂存区移除提出的文件修改

```
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   ch05.asciidoc


$ git reset HEAD ch05.asciidoc
Unstaged changes after reset:
M       ch05.asciidoc
```

另外，如果你只想取消暂存你对文件做出的某些修改，那么还可以使用 `reset` 命令和 `--patch` 参数。

撤销工作将会在第 6 章中更详细地介绍。

5.4.4 编写扩展提交消息

到现在为止，你可能都在编写精简的单行提交消息。如果你只是在练习版本控制命令，这么做没有关系，但如果你以后需要弄清楚一条叫“糟糕，再试一次”的提交消息是什么意思，那么恐怕不会很高兴。

我曾经将 Git 当成一个保存工作而不是记录结果的地方，我花了好长一段时间来摆脱这种思维方式。当我第一次开始使用版本控制时，为了利用这套先进的工具，我创建的提交粒度非常细（详见第 6 章）。这是因为我脑中想的是保存工作并撤销错误。用*保存*的想法思考时，我会想到点击保存按钮，或者使用 Control-Z 来撤销我刚输入的一些东西。当提交变得如此细小时，提交消息就没有什么意义了（“去吃午饭了”“试了一下”“还是不行”“糟糕”“测试”）。如果我想要回滚历史，那么要如何用这些提交信息来找到出错前正常工作的代码？找到自己的节奏也需要花费不少时间。

你的提交消息应该总是包含做出这个修改的原因，以及对你做出的修改的一个简要总结。为了编写一条详细的提交消息，对于你一直使用至今的单行短消息样式，你需要编写多行这样的消息。我通常使用下面的两步过程来编写我的提交消息（例 5-21）。

- 使用一个简短的单行消息将修改提交至仓库。
- 修补这个提交，完整地说明我在做出这个修改时在想什么。

例 5-21　编写详细的提交消息

```
$ git add --all
$ git commit -m "CH05: Adding technical edits."
$ git commit --amend
```

你不需要分成两步完成；你可以在第一次提交时忽略 -m 参数，直接跳转到消息编辑器，如下所示。

```
$ git commit
```

此时你的默认编辑器会打开，然后会提示你添加一条新的提交消息。这条消息的第一行将用于显示 --oneline，所有以 # 开头的行将在最终的消息中进行注释。一旦你编辑好你的提交消息，就需要保存它，然后退出编辑器以完成提交。

> **使用默认编辑器 Vim**
>
> 默认情况下，你编写提交消息的编辑器是 Vim。因为我喜欢 Vim，所以接受了这个默认编辑器。如果你不喜欢 Vim，可以参照附录 C 中的信息来更换编辑器。你需要掌握以下键盘命令来帮助你使用 Vim。
>
> - i 让你从视图模式切换到插入模式。如果你需要开始输入提交消息，那么将需要这个命令。
> - *esc* 返回可视化模式。随后你可以使用箭头键移动到另一行。
> - :w 将文件写回磁盘并保存。
> - :q 退出编辑器，回到命令行。
>
> 你还可以链式使用这些命令。例如，在编写完你的提交消息后，你可以使用 *esc* :wq 保存并退出编辑器。

在第 6 章中你将会学到如何通过交互式变基将细粒度的提交压缩成一个完整的想法。

5.4.5　忽略文件

最终，你可能会遇到这样的情况，Git 不断将你想添加的文件加入到仓库。如果你使用 Mac，这可能是烦人的 .DS_Store 文件。如果你使用 Linux，这可能是文本编辑器的 .swp 文件。如果你在进行一个 Web 项目，它可能会包含从 Sass 编译出的 CSS 文件。

如果你知道你最喜爱的文本编辑器或 IDE 会制造临时文件，且这些文件与项目无关，那么你应该创建一个全局设置来忽略这些文件。

首先，运行下面的命令来告诉 Git“忽略”文件的列表存放在什么位置。

```
$ git config --global core.excludesfile ~/.gitignore
```

你现在可以更新这个文件，每行一个文件名。你可以使用确切的文件名，或者通配符（例如：*.swp 将会匹配所有以 .swp 结尾的文件）。你可以查看 gitignore.io（https://www.gitignore.io/），使用它提供的忽略文件列表作为有用的起点。

另外，你或许希望特定仓库忽略特定的文件或文件扩展名。在这种情况下，你的最佳选择是在仓库中添加一个额外的 .gitignore 文件。这样做附带的好处是，你的队友一定不会意外地将构建文件提交上来。

完成下面几步来自定义特定仓库中应该被忽略的文件。

(1) 在项目根目录创建一个名为 .gitignore 的文件。
(2) 每行一个文件名，写上所有你一定不希望 Git 添加到仓库中的文件。你可以使用确切的文件名或通配符（如 `*.swp`）。
(3) 使用 `add` 和 `commit` 命令将 .gitignore 文件添加到你的仓库。

包含以上扩展名的文件将永远不会被添加至你的仓库，即使你使用了 `--all` 参数。

5.5　使用标签

标签用于定位指定的提交。你可以将标签视为书签。很多时候我应该使用标签，却都没有使用。因此，我依赖我的提交消息来寻找仓库中的特定节点。你或许会发现使用标签是一个好习惯，因为它能帮助你轻松地找到时间线上的节点。

多人团队中的标签

在本章中，我们提到的都是私有仓库，其中没有与其他队友共享的分支。当你的分支没有进行共享时，没有理由限制你应该如何以及何时使用标签。想打标签时就打吧！然而，共享分支上的标签通常用于开发目的，且应该遵循对团队有利的约定。

标签只能被添加到指定提交。为了明确你想要打上标签的提交，你或许会想要同时使用 `log` 和 `show` 两个命令。`log` 命令将会给你仓库中所有提交的列表（例 5-22），而 `show` 命令将会显示每个提交的详细信息。

例 5-22　最近提交的快速列表

```
$ git log --oneline

fa04c30 Initial import
```

一旦你觉得自己找到了你想继续研究的提交，就可以通过添加提交 ID 来获得这个提交开头的详细提交消息（例 5-23）。要将输出限制到仅这个提交，请添加一个可选的参数 `--max-depth=`，后面跟着你想要显示的记录条数。

例 5-23　单个提交的日志详细信息

```
$ git log fa04c30 --max-depth=1

commit fa04c309e3bb8de33f77c54c1f6cc46dc520c2ca
Author: emmajane <emma@emmajane.net>
Date:   Sat Oct 25 12:44:39 2014 +0100

    Initial import
```

如果你想要了解这个提交对象的更多信息，可以使用 `show` 命令以文本形式列出该提交中发生的修改（当然，如果这是二进制文件，例如图片，那就没有什么用了）。

例 5-24　使用 show 命令显示单个提交的日志消息和文本 diff

```
$ git show fa04c30

commit fa04c309e3bb8de33f77c54c1f6cc46dc520c2ca
Author: emmajane <emma@emmajane.net>
Date:   Sat Oct 25 12:44:39 2014 +0100

    Initial import

diff --git a/ch05.asciidoc b/ch05.asciidoc
new file mode 100644
index 0000000..8f82732
--- /dev/null
+++ b/ch05.asciidoc
@@ -0,0 +1,867 @@
+
+=== Verifying Git
+
+Before we dive into using Git, you'll want to check and see which version is
installed. For our purposes, Gi

[etc]
```

一旦你找到了想要收藏的提交，就可以通过 `tag` 命令来完成这个操作。在例 5-25 中，为提交散列 `fa04c30` 创建了一个新的标签 `import`。

例 5-25　为某个提交对象添加一个新的标签 import

```
$ git tag import fa04c30
```

你现在可以使用不加任何参数的 `tag` 命令列出所有已有的标签（例 5-26）。

例 5-26　列出所有标签

```
$ git tag
```

屏幕上将会输出一列标签。到目前为止，我们只添加了一个标签，所以这个列表非常简短，如下所示。

```
import
```

一旦标签被创建，你就可以通过标签所在的位置查看这个提交（例 5-27）。

例 5-27　查看带标签的提交

```
$ git show import
```

正如你之前所见，`show` 命令将会显示该提交的日志消息和文本 diff。

5.6　连接远程仓库

在 Subversion 这样的集中式版本控制系统中，只有一份仓库的主副本，所有工作都被写入那份副本。当你提交时，信息立即被上传至中央仓库，并且对其他人可见。在 Git 这样的去集中式版本控制系统中，没有每个人共同使用的单一仓库。我们通常会约定其中一个仓

库副本享有特权（并被视为代码的官方来源）。

当你一个人工作时，远程仓库更多是作为你本地仓库的备份，因为除非你将修改放到远程仓库，否则什么都不会发生。远程仓库可以在不同的本地开发环境之间转移代码。例如，你可以同时在一个笔记本和一个台式机上工作。远程仓库是在不同地方工作，让代码流动的好办法，这样你甚至可以在切换了机器之后继续工作。

如果你完成了本章前面的练习，那么现在应该知道你有三个本地仓库：一个从 GitLab 仓库克隆下来的仓库，一个从下载的 .zip 压缩包创建的仓库，以及一个从空目录创建而来的第三个仓库。这些仓库都是本地的，你无法将你的工作共享给别人，因为它们没有与之相关的远程仓库（从压缩包创建的仓库和在本地初始化的仓库），或者你没有远程仓库的写入权限（你克隆的仓库）。

为了上传你的工作，你将需要在 GitLab 上创建一个新的项目，并将它与你的一个已有仓库关联起来。

5.6.1　创建新的项目

如果你还没有远程仓库，那么将需要在 GitLab.com（https://gitlab.com/）上注册一个账户（这是免费的）并登录你的账户。你也可以用 GitHub、Twitter 或 Google 账户登录。尽管你可以在其他代码托管系统中完成这些步骤，例如 GitHub，但 GitLab 是一个开源产品，如果你需要在防火墙后练习源代码控制，那么可以在上面免费托管你的项目。

(1) 登录你的 GitLab 账户并前往你的信息中心页面（https://gitlab.com/）。
(2) 从项目摘要选项卡中，点击 New project 按钮。
(3) 输入项目路径，如 `gitforteams`。其余字段使用默认值即可。
(4) 点击 Create project。你将会被重新定向到一个指导页面，告诉你如何上传你的仓库。

5.6.2　添加第二个远程连接

GitLab 为你提供了只要复制粘贴即可的指令，帮助你将仓库上传至这个平台。不过，你不会希望完成所有步骤。从你的新项目页面上，查看第二部分，创建一个新的仓库（例 5-28）。

例 5-28　在 GitLab 上创建一个新仓库

```
mkdir my-git-for-teams
cd my-git-for-teams
git init
touch README.md
git add README.md
git commit -m "first commit"
git remote add origin git@gitlab.com:emmajane/my-git-for-teams.git
git push -u origin master
```

如果你已经在本地创建过一个仓库，那么能否看到你应该从哪一步开始？（提示：与标签为 Push an existing Git repository 的部分相比。）你已经完成了 `git remote add origin` 这行之前的所有步骤。如果你希望从头开始创建一个新的仓库，遵循所有的指令即可，然而你

现在已经有了三个本地仓库！与其（再次）创建一个新的仓库，不如通过添加一个远程连接来将三个仓库中的一个上传至 GitLab 中的新项目。选择三个仓库中的哪一个没有关系，但你只能选择一个，因为每个项目对应着一个独立的仓库。

当你将一个远程连接加入到仓库中时，必须为它分配一个别名（例 5-29）。默认情况下，这个别名是 *origin*。你可以用任何你喜欢的东西命名，比如 *pickles*、*peanutbutter* 和 *kittens*，Git 并不关心你选择的名称。使用 *origin* 的优点是你可以复制、粘贴更多网上的教程；缺点是 *origin* 不一定符合语义，尤其是当你的仓库事实上是从本地起步时。除此之外，*origin* 在你克隆远程仓库时已经被占用了。为了连接为三个本地仓库创建的远程仓库，我使用了 *my_gitlab* 这个别名。

例 5-29　使用自定义的名称在本地仓库中添加远程连接

```
$ git remote add my_gitlab git@gitlab.com:emmajane/my-git-for-teams.git
```

直到我最终学会管理 Git 中的命名，我才真正开始理解所有部分是如何拼到一起的。例如，我通常将代码托管系统的名称作为将远程连接的别名。“Git 团队协作”仓库的本地副本有以下远程仓库：`github`、`gitlab` 和 `bitbucket`（例 5-30）。

确认已通过 `remote` 命令正确添加远程仓库，如例 5-30 所示。

例 5-30　列出连接至你当前仓库的远程仓库

```
$ git remote --verbose
```

如果你已经将这个远程连接分配给了你克隆的仓库，那么将会看到下列两对远程连接。

```
my_gitlab       git@gitlab.com:emmajane/my-git-for-teams.git (fetch)
my_gitlab       git@gitlab.com:emmajane/my-git-for-teams.git (push)
origin  git@gitlab.com:emmajane/gitforteams.git (fetch)
origin  git@gitlab.com:emmajane/gitforteams.git (push)
```

你现在可以将你的工作从任何分支推送到你的远程仓库。

5.6.3　推送你的更改

为了上传你的修改，你需要远程仓库的连接、发布到仓库的权限和你想要上传修改的分支名。在你第一次推送分支时，需要明确告诉 Git 将这些东西放在什么位置。如果你使用 `push` 来推送修改，它会告诉你下一步怎么做。

免去输入密码的麻烦

如果你还没有将你的 SSH 密钥添加到代码托管系统（参见附录 D），那么将需要在每次推送修改前输入你的用户名和密码。

例如，如果你正在使用 *1-process_notes* 分支，并且想将它推送至远程仓库（例 5-31），那么将会得到一条错误消息（例 5-32）。

例 5-31　使用 push 命令上传分支

```
$ git push
```

例 5-32　在没有上游分支时，你将会得到一条错误消息

```
fatal: The current branch 1-process_notes has no upstream branch.
To push the current branch and set the remote as upstream, use

    git push --set-upstream origin 1-process_notes
```

这条错误消息为我们提供了非常有用的信息，但可能并不完全正确。与其说是将你的分支上传到远程仓库 *origin*，不如说我们事实上想要使用新的远程仓库 *my_gitlab*（例 5-33）。

例 5-33　在上传本地分支时设置上游分支

```
$ git push --set-upstream my_gitlab 1-process_notes
```

这个命令将会上传你的分支并且准备好在未来使用。现在不论你什么时候使用这个分支，你都可以使用简单得多的命令 `git push` 来上传你的工作。通过设置上游连接，事实上你建立了本地分支副本和远程仓库之间的关联。这与 `--track` 签出一个远程分支的作用是相同的，除了在这个例子中，你一开始使用的是远程的副本并随后添加了一个跟踪的本地副本。

5.6.4　分支维护

一旦代码经过了完整的测试，你就会希望将这个工单分支并入 *master* 分支（例 5-34）并删除本地分支（例 5-35）和这个工单分支（例 5-36）的远程副本。在单人团队中，你不太可能需要处理合并冲突。合并冲突将在第 7 章中介绍。

例 5-34　将工单分支并入你的主分支

```
$ git checkout master
$ git merge 1-process_notes
```

如果需要执行真正的合并，而不只是快进历史记录，或许会弹出一个编辑器让你编写提交消息。一般我使用默认的消息。一旦工作被并入 *master* 分支，你同样应该将 *master* 分支推送到远程仓库，如下所示。

```
$ git push --set-upstream my_gitlab master
```

现在，修改已经被并入 *master* 分支，没有必要继续保留这个工单分支了。为了保持你的仓库整洁，你可以更进一步，现在删除这个工单分支（例 5-35）。

例 5-35　删除这个分支的本地副本

```
$ git branch --delete 1-process_notes
```

如果还有没有并入其他分支的修改，Git 会不断提醒你，因此你不需要（过于）担心丢失未保存的工作。

最后，你同样需要在远程仓库做一些清理工作。你还应该删除那些修改已并入 *master* 分支的远程分支（例 5-36）。

例 5-36　删除不再需要的远程分支

```
$ git push --delete my_gitlab 1-process_notes
```

当清理结束时，是时候为你的下一个新想法重复这个步骤了。

5.7　命令指南

表 5-1 列出了本章用到的所有命令。这些命令均为 shell 命令，使用方式如下所示。

表5-1：基本的shell命令

命令	用途
`cd ~`	转到你的 home 目录
`mkdir`	创建新目录
`cd` *`directory_name`*	转到指定目录
`ls -a`	在 OS X 和基于 Linux 的系统下列出隐藏文件
`dir`	在 Windows 下列出文件
`touch` *`file_name`*	使用指定名称创建新的空文件

表 5-2 列出了 Git 应用使用的子命令。在命令行中，它们总是以命令 `git` 开头。

表5-2：基本的Git命令

命令	用途
`git clone` *`URL`*	下载一份远程仓库的副本
`git init`	将当前目录转换成一个新的 Git 仓库
`git status`	获取仓库状态报告
`git add --all`	将所有修改过的文件和新文件添加至仓库的暂存区
`git commit -m` *`"message"`*	将所有暂存的文件提交至仓库
`git log`	查看项目历史
`git log --oneline`	查看压缩过的项目历史
`git branch --list`	列出所有本地分支
`git branch --all`	列出本地和远程分支
`git branch --remotes`	列出所有远程分支
`git checkout --track` *`remote_name/branch`*	创建远程分支的副本，在本地使用
`git checkout` *`branch`*	切换到另一个本地分支
`git checkout -b` *`branch branch_parent`*	从指定分支创建一个新分支
`git add` *`filename(s)`*	仅暂存并准备提交指定文件
`git add --patch` *`filename`*	仅暂存并准备提交部分文件
`git reset HEAD` *`filename`*	从暂存区移除提出的文件修改
`git commit --amend`	使用当前暂存的修改更新之前的提交，并提供一个新的提交消息
`git show` *`commit`*	输出某个提交的详细信息
`git tag tag` *`commit`*	为某个提交对象打上标签

（续）

命令	用途
`git tag`	列出所有标签
`git show` *tag*	输出所有带标签提交的详细信息
`git remote add` *remote_name URL*	创建一个指向远程仓库的引用
`git push`	将当前分支上的修改上传至远程仓库
`git remote --verbose`	列出所有可用远程连接中 `fetch` 和 `push` 命令使用的 URL
`git push --set-upstream` *remote_name branch_local branch_remote*	将本地分支的副本推送至远程服务器
`git merge` *branch*	将当前存储在另一分支的提交并入当前分支
`git push --delete` *remote_name branch_remote*	在远程服务器中移除指定名称的分支

5.8 小结

在本章中，你已经学会了如何在单人团队中使用 Git。下面是在本章中列出的最佳实践。

- 总是在开始工作前定义好你要做的事情。这将会帮助你决定分支的名称，以及你想要从哪个分支开始工作。
- 当你在自己的分支上进行修改时，可以将其中一些修改或所有修改添加至暂存区。这能够帮助你确保一个提交只包含相关的工作。
- 无论你是在本地新建仓库还是克隆一个仓库，你总是可以在代码托管系统上创建一个新的项目，然后通过在本地仓库中添加一个新的远程来上传你的工作。
- 清理任务应该在你完成每行工作后进行。你可以将工单分支并入主分支，然后删除分支的本地和远程副本。

在下一章中，你将会学到如何通过 Git 时光机回到从前，撤销你的工作以及修改你的提交历史。

第 6 章

回滚、还原、重置和变基

本章也称为“糟糕”的一章。毕竟好人也会遇上坏事。幸运的是，Git 可以让你穿越回去，撤销已经犯下的错误。根据错误的严重程度，Git 提供了不同的命令，轻则微调一条提交消息，重则抹除历史记录。错误的提交和移除通常都发生在个人仓库中，但你处理错误的方式将会影响到其他人与代码库的交互。确保你总是以最友好的方式处理问题，帮助你的团队更加高效地工作。

完成本章学习之后，你将具备以下技能。

- 修补一个提交，并加入新的工作
- 将一份文件恢复到之前的状态
- 将你的工作目录恢复到上次提交时的状态
- 还原之前作出的变更
- 使用变基重塑提交历史记录
- 从仓库中移除一份文件
- 移除某个分支上因为错误的合并引入的提交

在本章中，你将会学习容易被忽略、但影响巨大的使用技巧。放慢节奏，画图思考一下，在运行一系列命令后你期望出现什么样的结果。这样你可以选择正确的子命令和正确的参数。同时，这样做还有助于你在下次需要执行相同任务时回忆起这些信息。

喜欢通过视频教程学习的读者可以参考本书附带的系列视频，Collaborating with Git（http://shop.oreilly.com/product/0636920034872.do）。

6.1　最佳实践

在本章中，你将会学习如何操作你的仓库历史记录。尽管本书中的练习易于上手，但压力

和紧张总是难免的，你或许会感到恐慌，害怕丢失工作。深呼吸，没什么大不了的。如果你将工作提交到了仓库，只要愿意搜寻下去，那么（几乎）总是能够找到这些工作。在 Git 中很难将工作完全删除，但相对来说容易丢失工作后找不回来。因此，在你学会修补历史记录前，确保你有趁手的恢复工具来帮助你摆脱困境。

6.1.1 描述问题

在 Git 中有很多方法都可以撤销工作，不同的方法适用于不同的情形。为了选择正确的方法，你需要确切知道你想要修改什么，以及在修改后会发生什么变化。在我初学版本控制时，通常会快速画个草图来描述我想要完成的工作，确保自己会使用正确的命令来完成工作。图 6-1 显示了你应该了解的三个概念：工作目录（你的文件系统中可见的文件）、暂存区（在下一次 `commit` 后即将写入仓库的变更的索引）和仓库（储存文件并记录了文件的每次变更）。

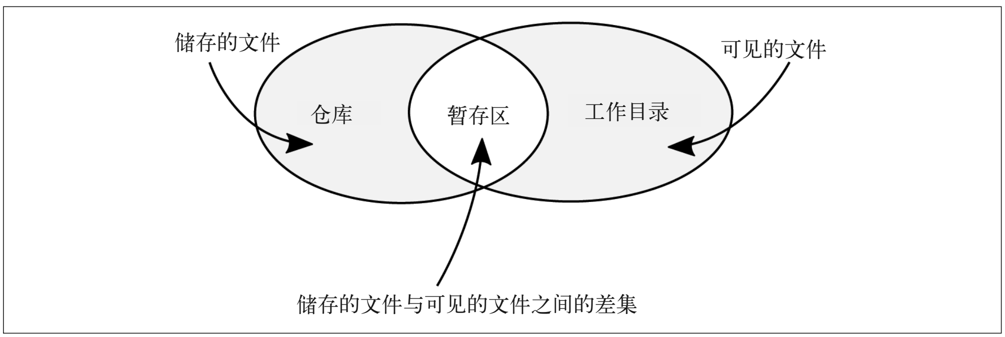
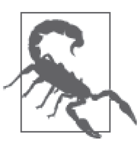


图 6-1：工作区、暂存区和仓库分别包含你的文件的不同信息



暂存区不会自动更新

图 6-1 中有些地方不那么准确，因为你需要使用 `add` 命令来显式地将工作放入暂存区，但对你来说，这是一个易于上手的工作模型。

Git 将信息存放在不同地方，当你将问题隔离在这些不同的地方时，将能够更好地选择正确的命令序列，将工作恢复到你想要的状态。表 6-1 总结了你在使用 Git 时可能遇到的一系列场景。

表6-1：选择正确的撤销方法

你想要……	备注	解决方案
舍弃工作目录中对一个文件的修改	修改的文件未被暂存或提交	`checkout --`*`filename`*
舍弃工作目录中所有未保存的变更	文件已暂存，但未被提交	`reset --hard`
合并与某个特定提交（但不含）之间的多个提交		`reset` *`commit`*
移除所有未保存的变更，包含未跟踪的文件	修改的文件未被提交	`clean -fd`

（续）

你想要……	备注	解决方案
移除所有已暂存的变更和在某个提交之前提交的工作，但不移除工作目录中的新文件		`reset --hard` *commit*
移除之前的工作，但完整保留提交历史记录（“前进式回滚”）	分支已经被发布，工作目录是干净的	`revert` *commit*
从分支历史记录中移除一个单独的提交	修改的文件已经被提交，工作目录是干净的，分支尚未进行发布	`rebase --interactive` *commit*
保留之前的工作，但与另一提交合并	选择 squash（压缩）选项	`rebase --interactive` *commit*

图 6-2 显示了第一种场景的图像。其他问题的答案可以在“Git 团队协作”网站（http://gitforteams.com/）上找到。

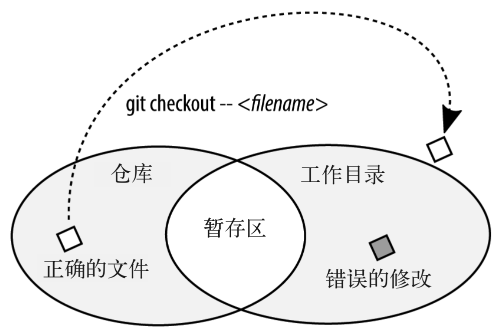


图 6-2：你希望舍弃工作目录中对一个文件的修改，错误的文件副本没有被暂存或提交

正如表 6-1 列出的示例，使用不同的参数会造成一些命令出现两种不同的结果。你可以在图 6-3 的流程图中找到你所处的场景。在纸上或电脑上重新画一遍这张图。重画一遍图通常比看一页书更容易记住，当你以后在 Git 中遇到这些选项时你会有更深的印象，还可以参考当时画的这张图。

你也可以写下你自己想要从中恢复的更改类型。创建一个列表，列出所有你想要从中恢复的问题场景。问题描述得越好，就越有可能找到正确的解决方案。在阅读本章时，你可以选择在图 6-3 中的流程图上扩展，或是创建你自己的图。记得在 Twitter 上用 #gitforteams（https://twitter.com/search?q=%23gitforteams&src=typd）标签分享你的工作。我期待看到你的想法！

6.1.2　使用分支进行试验性的工作

在分支树上，一个分支与它的兄弟分支是相互独立的。尽管它们可能有共同的祖先分支，你（通常）可以看到在树上移除一个分支却不会影响到其他分支。在 Git 中，你添加到仓库中的提交与一个或多个分支相关联。如果你签出一个不同的分支，并在这个新的分支上操作提交对象，那么系统会给这些提交对象分配一个新的标识符，而绑定到旧分支上的

提交对象不会改变。这也就是说，在新的私有分支中工作一定是安全的，当你对结果满意时，再将你的分支并入主分支（图 6-4）。

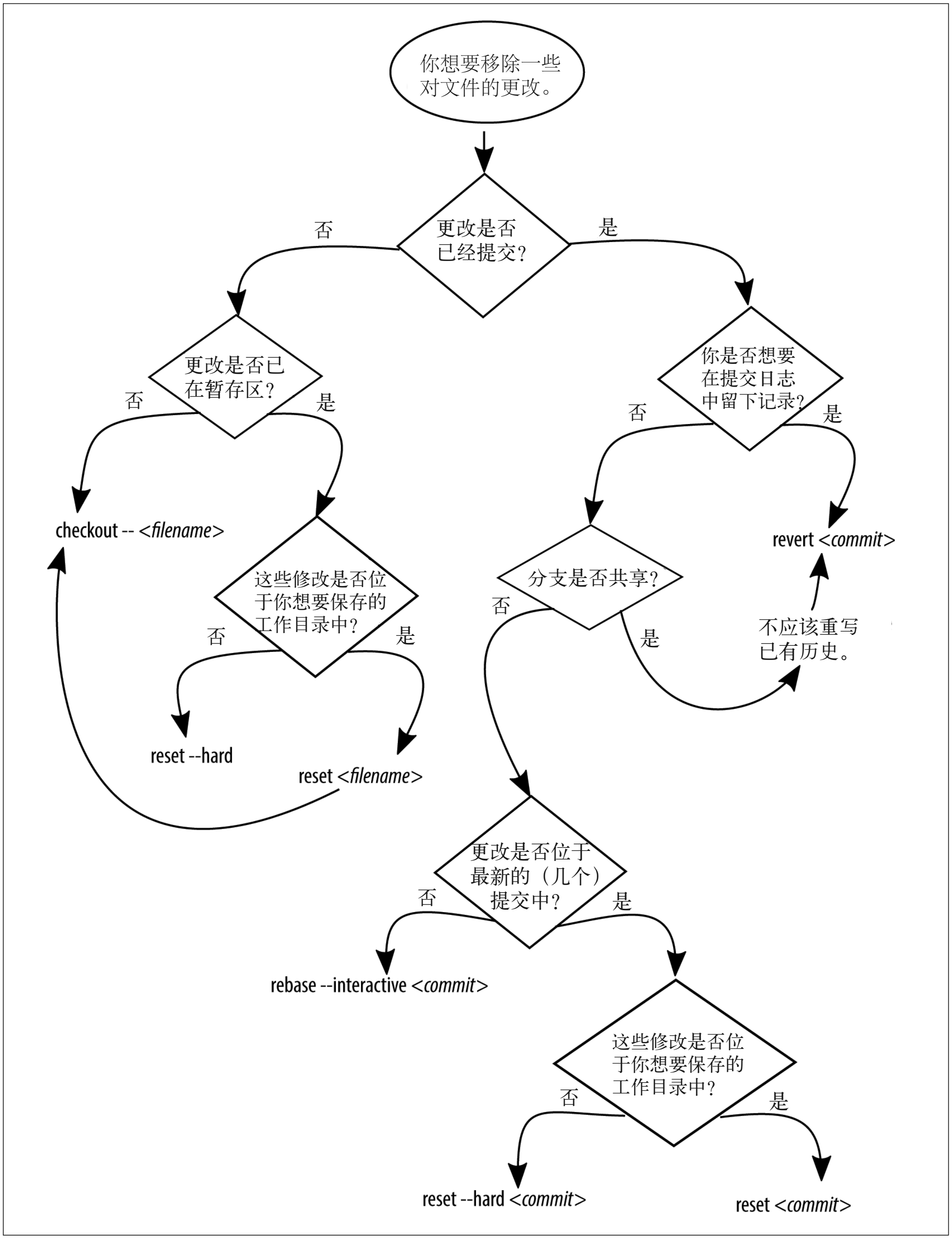


图 6-3：创建一个流程图来帮助你选择合适的命令

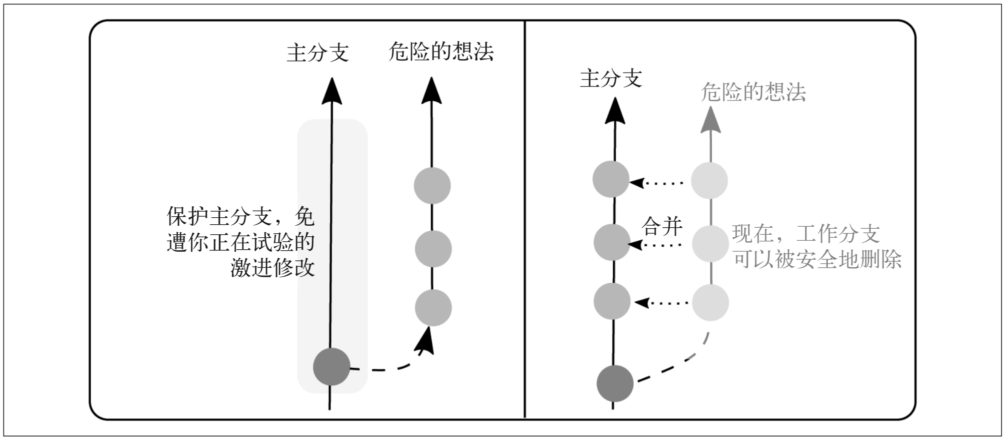


图 6-4：在分支中工作帮助你免遭意外的修改，在工作正确且完整后再将它并入主分支

之前，我们将工单作为创建和删除分支的起点。但如果你在一个工单上进行工作，但不确定应该使用两种方法中的哪一种。在这种情况下，你可以从你的工单分支上创建一个新的分支，在新的分支上进行试验性的修改（例 6-1），如果你想要保存这些修改，再将试验性的分支并入你的工单分支（例 6-2）。

例 6-1　使用试验性的分支来测试修改

```
$ git checkout -b experimental_idea
  (完成工作)
$ git add --all
$ git commit
```

在你的试验性分支上，你可能有一个或多个提交。当你合并两个分支时，可以选择使用带有 `--squash` 参数的合并，以将所有提交并入一个提交。通过这种方式合并分支，以后你将不能撤销分支合并。基于这个考虑，只有在合并你本来就不希望分离的分支时，才使用 `--squash`。

例 6-2　将你的试验性分支合并回主分支

```
$ git checkout master
$ git merge experimental_idea --squash

Squash commit -- not updating HEAD
Automatic merge went well; stopped before committing as requested

$ git commit
```

在合并后，你可以删除你的试验性分支（例 6-3）。

例 6-3　删除你的试验性分支

```
$ git branch --delete experimental_idea
```

如果你希望舍弃试验性的想法，完成最开始的步骤，省略把工作并入主工单分支这一步。要删除一个未合并的分支，你需要使用 `-D` 参数而不是 `--delete` 参数。

本章后续的小节中介绍了如何移除那些在你意识到它们只是试验性的修改之前就已经在分支上做出的提交。

6.2 分步变基

在 `rebase`、`reset` 和 `revert` 三个命令中，`rebase` 是唯一作用不局限于撤销工作的命令。一般来说，当我们谈论变基时，我们指的是使用父分支上的提交来更新一个分支的过程。通常这个过程非常直截了当：在你想要更新的分支上，运行 `rebase` 命令并使用父分支的名称作为参数。Git 将你的提交从工作所在的子分支上移除，并将父分支上产生的新提交添加到你的分支顶端。这样做看上去你的提交被放在了父分支中新增的修改后。打个比方，这就好像是 Git 吹着口哨假装什么都没有发生，但其实趁你不注意偷偷地把一盆花搬上了桌子。

尽管我们通常将变基称为“重演你的历史记录”，但它更准确的定义应该是回到之前，试着重新创建历史记录。如果你看过《回到未来》（或者现在类似的时空穿越片），你就会知道历史记录永远不可能与之前完全相同。`rebase` 也是如此。尽管看上去提交被转移到了一个新的分支的顶端，但事实上它们是全新的提交，有着自己的引用 ID。当你将这些提交应用到时间线上去时，如果新的历史记录和你尝试应用的工作发生了冲突，那么问题就来了。你将会处于分离式 `HEAD` 状态。晕了吗？还有另一种思考的方式：Git 允许我们重新讲述历史记录，我们可以随意插入新的事件。但是，事实上它不允许我们改变任何已经发生的事情。木已成舟，我们能做的只有改变我们讲述故事的方式。

在大多数时间里，在使用 `rebase` 命令更新分支时，这个过程非常迅速并且是自动完成的。但是，如果在变基过程中，你完成的工作与你试图放到父分支上的工作出现了有冲突的修改，这个过程将会停止，Git 会要求你在继续之前先手动解决冲突。这个冲突可能是文件内的修改，也可能是删除文件（其中一方删除了另一方编辑过的文件）。而 Git 只是一个简单的内容跟踪工具。如果总是由你这位专家给出合适的冲突解决方案，那么最终的产品质量将会更好。即使你希望 Git 能够自己解决，但让它停下来向你寻求帮助总是没错的。把这看作是人生中的一个宝贵的教训吧：向别人求助没什么大不了的。

`rebase` 造成困扰的第二个原因是当它被用于强制更新公共分支时。在这种情况下，时间线上将会出现两个（或多个）拥有不同 ID 的提交对象，它们包含相同的代码。为了帮助你选择应该采用变基还是合并，请参考变基、合并决策树（http://gitforteams.com/resources/merge-rebase.html）。

本节剩余内容描述在更新分支时遇到变基中冲突的处理流程。在我们的例子中，父分支（或源分支）的名称是 `master`，而我们尝试更新的分支（子分支）的名称是 `feature`。

6.2.1 开始变基

确保你父分支的本地副本与项目主仓库中最新的提交同步，如下所示。

```
$ git checkout master
$ git pull --rebase=preserve remote_nickname master
```

若有必要，请显式指定参数

在使用 `pull` 命令更新一个分支的本地副本时，参数中远程连接和远程分支的名称通常是可以忽略的。有些时候，如果一个仓库拥有多个远程连接，Git 有时会遗漏可用的更新。加上这两个额外的参数或许会有帮助。

当前分支上的修改与主项目不同步，而主项目中的新工作尚未被引入，如下所示。

```
$ git checkout feature
```

开始变基过程，如下所示。

```
$ git rebase master
```

如果没有冲突，Git 将会愉快地跳过这个过程并将你带到另一个终点，不需要你再进行任何附加操作。看到了吗？变基是很容易的！你应该试一试！但是，有些时候我们会遇到冲突……

6.2.2 文件删除造成的变基中冲突

当你和父分支中新的提交修改了同一行时，你在变基时会遇到冲突。作为一个简单的内容跟踪工具，Git 没法知道应该保留哪个修改，是我们的，还是他们的？与其胡乱猜一个，不如让 Git 停下来寻求你的帮助。事实上，我觉得 Git 这样做很贴心，把你当作了解这些内容的专家，而不是自以为是！只不过，这个过程称为“在分离式 HEAD 状态下解决冲突”，而不是“问你呢，专家！应该怎么办”。它的用语看上去很可怕，但其实这个过程非常友好。

为了解决某个冲突，你需要扮成专家的样子，帮助 Git 决定接下来做什么。

本节介绍了一个变基中冲突的例子。ch10.asciidoc 文件在源分支 `master` 中被删除了，但我在 `feature` 分支上继续对它进行修改。Git 在解决冲突的时候会遇到问题。我是想要保留这个文件，还是应该删除它？ Git 将我置于分离式 HEAD 状态中，因此我需要告诉 Git 我希望如何继续，如下所示。

```
First, rewinding head to replay your work on top of it...
Applying: CH10: Stub file added with notes copied from video recording lessons.
Using index info to reconstruct a base tree...
A       ch10.asciidoc
Falling back to patching base and 3-way merge...
CONFLICT (modify/delete): ch10.asciidoc deleted in HEAD and modified in CH10:
Stub file added with notes copied from video recording lessons.. Version CH10:
Stub file added with notes copied from video recording lessons. of ch10.asciidoc
left in tree.
Failed to merge in the changes.
Patch failed at 0001 CH10: Stub file added with notes copied from video
recording lessons.
The copy of the patch that failed is found in:
   /Users/emmajane/Git/1234000002182/.git/rebase-apply/patch

When you have resolved this problem, run "git rebase --continue".
```

```
If you prefer to skip this patch, run "git rebase --skip" instead.
To check out the original branch and stop rebasing, run "git rebase --abort".
```

与这个输出相关的信息如下所示。

```
When you have resolved this problem, run "git rebase --continue".
```

此信息告诉我需要按以下步骤操作。

(1) 解决合并冲突。
(2) 当我认为合并冲突已经被解决时，运行下面的命令。

```
git rebase --continue
```

使用我喜欢的文件对比工具打开有问题的文件，完成第 1 步。

```
$ git mergetool ch10.asciidoc
```

文件中不再显示有合并冲突，因此我退出合并工具，并继续进行 Git 提示的下一步操作，如下所示。

```
$ git rebase --continue
```

Git 返回了下面的消息。

```
ch10.asciidoc: needs merge
You must edit all merge conflicts and then
mark them as resolved using git add
```

这个消息可没什么帮助！我查看了那个文件，发现没有合并冲突。我使用 `status` 命令询问 Git 出了什么问题，如下所示。

```
$ git status
```

Git 输出了如下内容。

```
rebase in progress; onto 6ef4edb
You are currently rebasing branch 'ch10' on '6ef4edb'.
  (fix conflicts and then run "git rebase --continue")
  (use "git rebase --skip" to skip this patch)
  (use "git rebase --abort" to check out the original branch)

Unmerged paths:
  (use "git reset HEAD <file>..." to unstage)
  (use "git add/rm <file>..." as appropriate to mark resolution)

        deleted by us:   ch10.asciidoc

no changes added to commit (use "git add" and/or "git commit -a")
```

啊！我发现了两条线索。`Unmerged paths` 和后面的 `deleted by us: ch10.asciidoc` 这两段文字。好吧，我不希望这个文件被删除。这个信息很有用，因为 Git 告诉我 `deleted by us`，而我知道我不希望删除这个文件，因此我需要 `unstage`（取消暂存）Git 的修改。取消暂存一个修改等于是告诉 Git"你打算做什么？别那么做。事实上，忘掉你对这个文件的所有想法。Git，重置你的 `HEAD`（头指针）"。

Git 通过下面的文字，告诉我如何阻止这个变更发生。

```
(use "git reset HEAD <file>..." to unstage)
```

根据这个消息的指导，运行下面的命令。

```
$ git reset HEAD ch10.asciidoc
```

现在，这个命令所做的事情其实是清除暂存区，将指针回退到最新的一个已知的提交。由于正陷于变基过程中，处在分离式 HEAD 状态而不是在一个分支上，因此 reset 清除了暂存区，并且回到了变基前的最新状态。在这种情况下，我回到了文件的旧版本，这也正是我所希望的。当我继续变基时，我会使用 featuer 分支上最新的版本来替换这个文件内容。如果我希望保留他们对文件的删除，我应该跳过这一步，根据指示进入下一步，通过后文所述的方式将文件添加到暂存区。

在替换了我那个章节的文件之后，让我们看看 Git 给出了怎样的线索让我继续，如下所示。

```
$ git status
```

Git 的输出如下所示。

```
rebase in progress; onto 6ef4edb
You are currently rebasing branch 'ch10' on '6ef4edb'.
  (all conflicts fixed: run "git rebase --continue")

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        ch10.asciidoc

nothing added to commit but untracked files present (use "git add" to track)
```

这么看来我仍然拥有这份文件（这很棒！），但 Git 仍然感到很困惑，不知道要怎么做，因为据它所知，这份文件应该已经被删除了。我需要显式地将这份文件加回到仓库中，Git 告诉我应该按下面这样做。

```
Untracked files: (use "git add <file>..." to include in what will be
committed) ch10.asciidoc
```

如果只有一份文件受到影响，这个格式看上去有些奇怪。但在这个例子中我们有长长的一列文件，格式清晰易懂。

根据 Git 的请求，我现在将 ch10.asciidoc 文件加入暂存区，如下所示。

```
$ git add ch10.asciidoc
```

到现在，我知道 add 命令只是一个流程的开始，我还需要 commit 文件，但在变基中这个规则有所差别。我再次查看 status 命令的输出，询问 Git 接下来该怎么做，如下所示。

```
$ git status
```

Git 的输出如下信息。

```
rebase in progress; onto 6ef4edb
```

```
You are currently rebasing branch 'ch10' on '6ef4edb'.
  (all conflicts fixed: run "git rebase --continue")

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        new file:   ch10.asciidoc
```

好，这个消息的意思是还有要提交的变更（嗯，早就知道了），但它没有让我提交这些变更。相反，它在下面这条消息中让我继续变基。

```
all conflicts fixed: run "git rebase --continue"
```

我使用这个命令继续，即使我们通常搭配使用 `add` 和 `commit` 来保存修改，如下所示。

```
$ git rebase --continue
```

6.2.3　单个文件合并冲突造成的变基中冲突

在重新进入变基过程后，Git 在重演提交时又遇到了另一个冲突。输出如下所示。

```
Applying: CH10: Stub file added with notes copied from video recording lessons.
Applying: TOC: Adding Chapter 10 to the book build.
Using index info to reconstruct a base tree...
M       book.asciidoc
Falling back to patching base and 3-way merge...
Auto-merging book.asciidoc
CONFLICT (content): Merge conflict in book.asciidoc
Recorded preimage for 'book.asciidoc'
Failed to merge in the changes.
Patch failed at 0002 TOC: Adding Chapter 10 to the book build.
The copy of the patch that failed is found in:
   /Users/emmajane/Git/1234000002182/.git/rebase-apply/patch

When you have resolved this problem, run "git rebase --continue".
If you prefer to skip this patch, run "git rebase --skip" instead.
To check out the original branch and stop rebasing, run "git rebase --abort".
```

又是一个冲突。Git 好麻烦！难怪人们都在抱怨变基！好吧，至少这次是另一个文件了（`CONFLICT (content): Merge conflict in book.asciidoc`）。我再次仔细观察了 `status` 命令的输出，看看 Git 有没有给我其他线索。

```
$ git status
```

Git 返回的输出如下所示。

```
rebase in progress; onto 6ef4edb
You are currently rebasing branch 'ch10' on '6ef4edb'.
  (fix conflicts and then run "git rebase --continue")
  (use "git rebase --skip" to skip this patch)
  (use "git rebase --abort" to check out the original branch)

Unmerged paths:
  (use "git reset HEAD <file>..." to unstage)
```

```
  (use "git add <file>..." to mark resolution)

        both modified:   book.asciidoc

no changes added to commit (use "git add" and/or "git commit -a")
```

长叹一口气。好吧，Git。让我们来看看这个文件中有什么冲突。

```
$ git mergetool book.asciidoc
```

使用我最喜欢的合并工具打开这个文件，我看到这个文件中确实有一个合并冲突。这个合并冲突标记显示为三行。一行显示两个合并的分支，一行显示应该如何解决合并冲突。我选择我想要保留的补丁片段来解决这个冲突。保存文件，关闭合并工具，再次使用 `status` 命令询问 Git 是否满意，如下所示。

```
$ git status
```

Git 返回的输出如下所示。

```
rebase in progress; onto 6ef4edb
You are currently rebasing branch 'ch10' on '6ef4edb'.
  (fix conflicts and then run "git rebase --continue")
  (use "git rebase --skip" to skip this patch)
  (use "git rebase --abort" to check out the original branch)

Unmerged paths:
  (use "git reset HEAD <file>..." to unstage)
  (use "git add <file>..." to mark resolution)

        both modified:   book.asciidoc

no changes added to commit (use "git add" and/or "git commit -a")
```

这个消息有些误导人，因为我已经修复了冲突。此时，我打开文件再确认一次。好，没有冲突。因此，我继续阅读下一组指示：`use`“`git add <file> ...`”`to mark resolution` 和 `both modified: book.asciidoc`，如下所示。

```
$ git add book.asciidoc
```

再次检查 `status`，如下所示。

```
$ git status
```

Git 返回的输出如下所示。

```
rebase in progress; onto 6ef4edb
You are currently rebasing branch 'ch10' on '6ef4edb'.
  (all conflicts fixed: run "git rebase --continue")

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   book.asciidoc
```

和之前一样，我没有一起使用 `add` 命令和 `commit` 命令。相反，Git 给了我如下的指示：`all`

`conflicts fixed: run“git rebase --continue”`，因此我继续变基的流程，如下所示。

```
$ git rebase --continue
```

Git 的输出如下所示。

```
Applying: TOC: Adding Chapter 10 to the book build.
Recorded resolution for 'book.asciidoc'.
Applying: CH10: Outline of GitHub topics
```

变基的流程已经全部完成。我的 *feature* 分支的副本已是最新，包含之前提交到 *master* 分支上的变更。

在变基时，你有很多方式解决冲突。慢慢来，仔细阅读指示。如果你没有获得什么有用的信息，试试 `status` 命令，看看 Git 有没有给你什么更有用的信息。如果你对眼前发生的事情感到十分恐慌，那么总是可以使用 `git rebase --abort` 命令退出这个流程。你将回到开始变基前你的分支所处的状态。

6.3 定位丢失的工作概述

在 Git 中，完全舍弃某个提交过的工作是非常困难的。但是，弄丢你的工作是很容易的事，就像我弄丢钥匙，弄丢眼镜，弄丢钱包和用尽家人的耐心一样。如果你认为你丢失了一些工作，那么首先应该做的事情是找到保存之前工作的提交。`log` 命令显示了某个分支上的提交，`reflog` 命令列出了在仓库的本地副本中的完整历史记录。也就是说你在从远程服务器克隆下来的仓库中工作时，reflog 历史记录起始于你把仓库从仓库克隆到本地环境时的那一刻，而 log 历史记录将会显示使用 `init` 命令创建仓库后的所有提交消息。

如果你还没有一个仓库，下载一份本书的项目仓库，比较 `reflog` 和 `log` 两个命令的输出（例 6-4）。

例 6-4　比较 `log` 和 `reflog` 的输出

```
$ git clone https://gitlab.com/gitforteams/gitforteams.git

Cloning into 'gitforteams'...
remote: Counting objects: 1084, done.
remote: Total 1084 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (1084/1084), 12.07 MiB | 813.00 KiB/s, done.
Resolving deltas: 100% (628/628), done.
Checking connectivity... done.

$ git log --oneline

e8d6aff Updating diagram: Adding commit ID reference to rebase.
ae56a1f Adding workflow diagram for: reset, revert, rebase, checkout.
2480520 Merge pull request #5 from xrmxrm/1-markdown_fixes
ee46470 Fix some markdown Issue #1

$ git reflog

2f17715 HEAD@{1}: clone: from https://gitlab.com/gitforteams/gitforteams.git
```

如果你刚刚克隆完仓库，那么在引用日志中只会看到一行。当你进行了更多操作后，引用日志将会逐渐增加。下面是本书仓库的一份样例输出。

```
fdd19dc HEAD@{157}: merge drafts: Fast-forward
af9e2c8 HEAD@{158}: checkout: moving from drafts to master
fdd19dc HEAD@{159}: merge ch04: Merge made by the 'recursive' strategy.
af9e2c8 HEAD@{160}: checkout: moving from ch04 to drafts
e296faa HEAD@{161}: commit (amend): CH04: first draft complete
dd87941 HEAD@{162}: commit: CH04: first draft complete
```

这个历史记录是私有的。谢天谢地，只有你可以看到它！它包含你所做的所有事，包括那些并不影响代码的事情，比如签出一个分支。

log 和 reflog 这两个命令都会显示仓库中某个状态的提交 ID。只要你找到了提交 ID，你就可以签出这个提交（例 6-5），及时将代码库的版本临时恢复到那个节点。

例 6-5　在仓库中签出特定提交

```
$ git checkout commit

Checking out files: 100% (2979/2979), done.
Note: checking out 'a94b4c4'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b new_branch_name

HEAD is now at a94b4c4... Fixing broken URL to the slides from the main README file.
Was missing the end round bracket.
```

当你签出一个提交时，你将与某个分支相连的历史记录分离。不过，这并没有听上去那么吓人。一般情况下，当我们在 Git 中工作时，我们使用线性的方式来展现历史记录。当我们签出一个提交时，我们处于暂停状态（图 6-5）。

这时人们通常开始恐慌，简单地说，就是你的头指针处于分离状态下！遵循 Git 提供的指示将会帮你回到正确的状态。如果你希望保存你所在的状态，创建一个新的分支，然后你的状态将会记录在新的分支上，如下所示。

```
$ git checkout -b restoring_old_commit
```

此时，如果你有什么忘了添加的东西（或者你想要删除不再相关的旧工作），你可以在新的分支中继续添加一些修复。在你完成后，你需要决定如何将新的分支合并回工作分支。你可以选择将新的分支 merge（合并）到已有的分支上，或者只是 cherry-pick（拣选）一些想要保存的提交。让我们从合并开始，因为你在第 5 章中应该已经熟悉了这个方式，如下所示。

```
$ git checkout working_branch
$ git merge restoring_old_commit
```

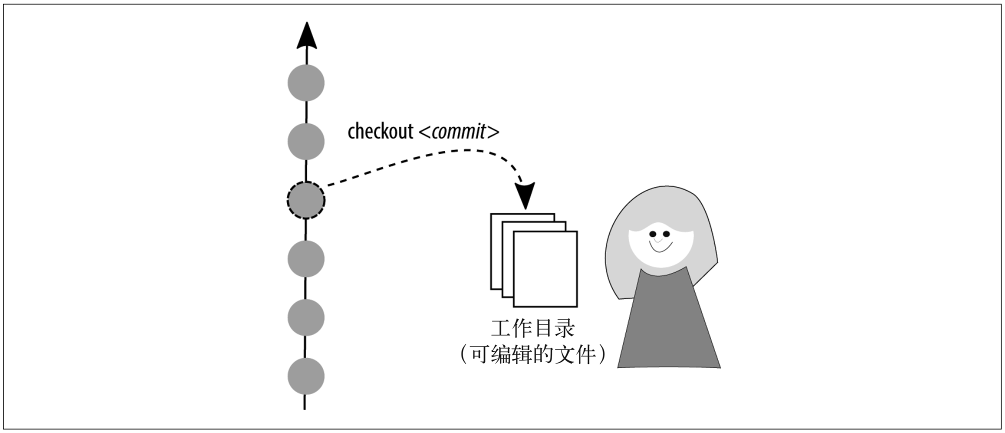


图 6-5：在分离式头指针状态下，你与分支的线性历史记录暂时失去连接

在合并完成后，你现在应该删除临时分支来整理你的本地仓库，如下所示。

```
$ git branch --delete restoring_old_commit
```

如果你已经发布了临时分支，并且希望从远程仓库删除它，那么你需要显式进行这一步，如下所示。

```
$ git push --delete restoring_old_commit
```

如果临时分支上包含很多无关的工作，这种方法可能会造成巨大的混乱。在这种情况下，使用 `cherry-pick`（拣选）命令可能更为合适（例 6-6）。这个命令有多种使用方式——使用 `git help cherry-pick` 命令查看它的文档。我倾向于使用我想要复制到当前分支的提交 ID。可选的 `-x` 参数在提交消息后增加了一行，告诉你这个提交是从别处拣选的，而不是在当前分支上原创的。新增的这一行使得这个提交在以后更好辨认。

例 6-6　使用 cherry-pick 将提交复制到新的分支

```
$ git cherry-pick -x commit
```

假设这些提交被干净地应用到了你当前的分支，你将看到如下所示的一条消息。

```
[master 6b60f9c] Adding office hours reminder.
 Date: Tue Jul 22 08:36:54 2014 -0700
 1 file changed, 2 insertions(+)
```

如果出现了问题，你可能需要手动解决提交冲突。输出将会如下所示。

```
error: could not apply 9d7fbf3... Lesson 9: Removing lesson stubs from
subsequent lessons.
hint: after resolving the conflicts, mark the corrected paths
hint: with 'git add <paths>' or 'git rm <paths>'
hint: and commit the result with 'git commit'
```

合并冲突在第 7 章中将会更详细地介绍。如果你在拣选提交时遇到了冲突，请直接跳到那一章。

你可能会遇到的另一个输出是：你想要并入的提交是一个合并提交。在这种情况下，你将会需要选择父分支。你可以通过试图拣选一个提交时 Git 给出的如下输出来辨别这个情况。

```
error: Commit 0075f7eda6 is a merge but no -m option was given.
fatal: cherry-pick failed
```

确认你想要保留的父分支是图形化输出的日志中左侧第一个分支（从左到右数），如下所示。

```
$ git log --oneline --graph
```

然后，再次运行 `cherry-pick` 命令，这次使用 `--mainline` 参数标识父分支，如下所示。

```
$ git cherry-pick -x commit --mainline 1
```

最后，如果你决定不保留恢复的工作，可以舍弃这些变更，如下所示。

```
$ git reset --merge ORIG_HEAD
```

公共的历史记录不应该被修改

`reset` 命令不应该被用于共享的分支来移除已经被发布的提交。撤销共享分支上的变更将在本章后续内容中提到。

如果你练过了本节中的每个例子，你现在应该可以签出一个提交，创建一个新的分支来从分离式 HEAD 状态中恢复，将一个分支中的变更合并到另一个分支，将提交拣选到一个分支并删除本地分支。

6.4　还原文件

如果你在工作时不小心错误地删除了文件，而事实上你希望保留这个文件。或者可能你编辑了一个不该编辑的文件。在这些修改被锁定（或*提交*）前，你可以*签出*这些文件。你可以将文件内容恢复到你当前所在的分支中最新的已知提交，并储存这个文件，如下所示。

```
$ rm README.md
$ git status

On branch master
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        deleted:    README.md

no changes added to commit (use "git add" and/or "git commit -a")
```

状态消息说明了如何撤销变更并恢复被删除的文件，如下所示。

```
$ git checkout -- README.md
```

如果你已经暂存了文件，你需要使用 reset 命令在恢复文件之前将它取消暂存。为了试验这一点，你需要首先删除一个文件，然后使用 add 命令将这些修改添加到暂存区，最后使用 status 命令来验证你的下一步操作，如下所示。

```
$ rm README.md
$ git add README.md

$ git status

On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        deleted:    README.md
```

此时，你之前使用过的 checkout 命令将不再有用。相反，遵循 Git 给出的指示来取消暂存你想要恢复的文件，使用 Git 的 HEAD（头指针）快捷方式，也就是指向当前分支最近一次提交的指针，如下所示。

```
$ git reset HEAD README.md
```

一旦文件被取消暂存后，你可以像之前使用 checkout 命令那样恢复被删除的文件，如下所示。

```
$ git checkout -- README.md
```

如果你愿意，可以把上面两个命令合并成一个，如下所示。

```
$ git reset --hard HEAD -- README.md
```

如果你想要撤销工作目录中的所有变更，将所有文件恢复到上一个提交的版本，你不需要每次更改一个文件。你可以批量完成这个操作，如下所示。

```
$ git reset --hard HEAD
```

现在，你应该可以恢复工作目录中被删除的文件了。

6.5　使用提交

提交是你的仓库中某一时刻的一份快照，包含仓库中所有文件的状态。其中的每个提交都可以在你的历史记录中操作。你可以使用 reset 命令完全移除这个提交，你可以使用 revert 命令反转一个提交的影响（但仍在历史记录中保留这个提交），你可以使用 rebase 命令修改提交的顺序。如果你已经发布了这些提交，绝对不要改变你的仓库历史记录。这是因为只要稍加修改就会导致它以一个新的提交 SHA 储存在仓库中，即使代码与分支顶端的一模一样。这是因为 Git 假设所有新的提交 ID 包含必须记入的新信息，不管这些提交中的文件内容是否改变。

在本节中，我们假设你正在操作的提交尚未共享给其他人（也就是说，你还没有推送你的分支）。我们会单独介绍变更共享分支的历史记录的方法。

6.5.1　修补提交

如果你意识到你刚刚创建的提交还需要一点点修改，你可以修补（amend）这个提交来引入更多文件或者更新这个提交的说明。我频繁地使用这个命令来将我简练的单行提交消息转换成格式良好的工作总结。

不要改变共享的历史记录

如果你已经推送了工作，回到过去“修复”共享历史记录是一个不好的习惯。

如果你在工作目录中改变了文件，你需要在修补提交前将这些文件添加（`add`）到暂存区（例 6-7）。如果你只是在更新提交消息，而没有新的或修改过的文件需要添加，你可以忽略 `add` 命令，然后直接跳到 `commit` 命令。

例 6-7　使用 `--amend` 更新之前的提交

```
$ git add --all
$ git commit --amend
```

你的新变更将会被添加到之前的提交，并且 Git 会为这个修改过的提交对象分配一个新的 ID。

还有更多的提交选项

构建你的提交对象的方法不止这些。我列出了我最常使用的一些选项。你可以阅读 `commit` 的相应教程页来寻找其他用法。你可以通过运行下面的命令来访问这些信息：`git help commit`。

不过，如果你希望修补更多提交，你将会用到 `reset` 或 `rebase`。

6.5.2　使用reset合并提交

`reset` 命令在撤销的流程中扮演了很多角色。在这个例子中，我们会使用它来模拟变基时 `squash`（压缩）的作用。对 `reset` 作用最基本的解释是，它其实只是修改了头指针的指向。不管你用手指着哪个提交，Git 都会把它当作你的分支当前的 `HEAD`（或顶端）。

`reset` 会改变你已记录的历史记录

`reset` 将会改变你的历史记录，因为它移除了指向提交的引用。如果有人试图合并他们旧版本的分支，他们将会重新引入你尝试移除的提交。因此，最好只在尚未与他人共享的分支上使用 `reset` 来改变分支历史记录（也就是说你本地创建的分支，且你尚未将它推送到服务器）。

之前你使用 `reset` 命令在提交前取消暂存工作。这次你将使用 `reset` 来移除你的分支历史记录中的提交对象。想象一串珠子。假设串上有 20 粒珠子。握住第四颗珠子，让前三个从串上滑下。你现在有一串短了一些的珠子和三个散落的珠子。你在使用 `reset` 命令时添

加的参数决定了这些珠子的命运。

如果你希望舍弃你移除的提交对象中的内容，你需要使用 hard 模式的 reset。这个模式通过 --hard 参数启用。当你使用 hard 模式时，提交对象将被移除，工作目录也将被更新，那些提交对象中储存的内容同样会被移除。如果你在重置你的工作时没有使用 --hard 选项，Git 保留工作目录中的内容不变，但将提交对象扔回原点。这个效果等同于将之前的提交中的所有修改都写进了一份巨大的工作里。这些工作正在等待被添加并提交。

reset 重建了分支顶端

有段时间，我脑中一直以为 reset 是用来撤销一个指定提交的操作。这个定义对于 revert 命令来说是正确的，而不是 reset。reset 命令将当前分支的最新提交重设为指定提交。或许如果命名为“restore（恢复）”、“promote（提升）”或甚至“set（设为）”，我的头脑将能够更好地区分这两个命令。记住：reset 的对象是要保留的东西，而 revert 的对象是要舍弃的东西。

继续使用我们的珠子比喻，假设你希望重置你的珠串，把最新的三颗珠子替换成一颗大珠子。你用 reset 命令将珠串新的末端指向距离末端第四颗珠子。然后，你将这三颗珠子从串上摘下。（如果使用了 --hard 参数，这些珠子将被丢弃。）相反，我们将要重塑这些珠子，然后将它们作为一个新的提交放回到串上。

提交必须是连续的，并以最新的提交结束

为了让这个操作生效，你需要连续地压缩提交，直到最新的提交。我们正在做的事其实是交互式变基的基础。通过 reset 的这种使用方式，你将被限制在最新的几个提交中。通过变基，你将能够选择任意范围内的提交。

使用 log 命令，辨认你想要保留的最新提交。这个提交将会成为分支的新顶端，如下所示。

```
$ git log --oneline

699d8e0 More editing second file
eabb4cc Editing the second file
d955e17 Adding second file
eppb98c Editing the first file
ee3e63c Adding first file
```

还是回到我们的三个珠子的比喻，我想要保留的珠子，也就是我的项链的顶端是 eppb98c。（它是从末端数第四颗珠子，如果你的注意力完全集中于移除三颗珠子上，可能不是那么符合你的直觉。）我们将手指放在我们想要保留的珠子上，让剩下的珠子从串上滑落下来，如下所示。

```
$ git reset eppb98c
```

现在我们有三个散落的珠子四处乱撞。这些珠子在我们的仓库中称为*未跟踪的变更*。文件的内容将不会被改变。

你可以使用 diff 命令来查看你的新提交中包含的内容，如下所示。

```
$ git diff
```

为了将这三个提交中做出的编辑合并到一个单独的提交，使用 add 命令捕捉暂存区中的所有变更，如下所示。

```
$ git add --all
```

确保这些文件现在都被暂存并准备好保存，如下所示。

```
$ git status
```

现在这些文件已经被暂存，diff 命令不再向你显示你将提交到仓库的内容。因此，你需要检验被暂存的这些变更，如下所示。

```
$ git diff --staged
```

> **暂存也是一种缓存**
>
> staged 参数是 --cached 参数的别名。我选择使用别名的版本，因为它和我提到暂存变更时使用的术语更加契合。如果你搜索这个参数的更多资料，一定要去看一看它原本的参数名。

一旦你对新提交的内容感到满意，你可以继续下一步，完成这个提交流程，如下所示。

```
$ git commit -m "Replacing three small beads with this single, giant bead."
```

现在，这三个提交将被合并成一个单独的提交。

如果你无法完全理解 reset 这个词，以及为什么要回到目标提交的前一个提交，我鼓励你使用相对历史记录而不是提交 ID。例如，如果你希望将你的分支中的三个提交压缩成一个，应该使用下面的命令。

```
$ git reset HEAD~3
```

这个版本的命令将你的仓库置于上一个例子中的状态，不过指针看上去使用了另一种描述。两种方法都行。选择任何一种你喜欢的描述。我个人认为，我想要重置的提交够多了，因此使用提交 ID 而不是向前数数会更容易。

如果你完成了本节中的所有例子，你现在应该已经能够恢复一个删除的文件，并将多个更小的提交合并成一个。

6.5.3　使用交互式变基修改提交

变基这样的主题既吸引了众多狂热的支持者，也有着强烈的反对者。尽管我在使用这个命令时没有遇到技术问题，我大方地承认我不喜欢它的工作方式。变基主要用于更改历史记录的记录方式，通常保留你的工作目录中的文件内容不变。若是被错误地使用，它可能会造成共享分支的混乱，其中不同 ID 的新提交对象储存了相同的工作。但我的抱怨更多是关于这个想法：可以根据你的喜好随意地重写历史记录。在软件之外的世界里，历史记录否定主义（https://en.wikipedia.org/wiki/Historical_revisionism_(negationism)）是错误的。

抛开这些抱怨，变基是 Git 选定的模型，因此它与很多工作流配合良好。（我在合适时用到它，即使是在我的个人项目中，没有外部团队强制要求我这么做。）使用变基的一个合适的场景是，保持一个分支最新（正如第 3 章和 6.2 节中讨论过的那样）；第二个场景是在发布工作前，交互式变基允许你将提交安排成易于阅读的历史记录。在本节中，你将学到这两种方法中的后一种。

如果你一直都提交细小的想法，那么交互式变基会尤为有用，只在你的历史记录中保留包含一部分想法的提交。如果你有很多提交，因为同行评审或是你自己回头一想，觉得之前作出的决定不是正确的方法。清理你的历史记录，只保留良好、有意义的提交，这样你能够更容易使用第 9 章即将介绍的 `bisect` 命令。为了帮助你理解这个概念，我创建了一个简单的动画（http://gitforteams.com/assets/animation-rebasing.svg），展示将几个小的提交压缩成一个完整的想法的基本原则。

你首先要做的是选择项目中你想作为起点的一个提交，（我通常选择比我认为我需要的提交更早一个的提交，以防万一）。假设你的分支历史记录中包含下列提交。

```
d1dc647 Revert "Adding office hours reminder."
50605a1 Correcting joke about horses and baths.
eed5023 Joke: What goes 'ha ha bonk'?
77c00e2 Adding an Easter egg of bad jokes.
0f187d8 Added information about additional people to be thanked.
c546720 Adding office hours reminder.
3184b5d Switching back to BADCamp version of the deck.
bd5c178 Added feedback request; formatting updates to pro-con lists
876e951 Removing feedback request; added Twitter handle.
```

你决定与笑话相关的三个提交应该被压缩成一个单独的提交。查看这个提交的上一个提交，选择 `0f187d8` 作为你的起点。你现在已经准备好进入变基的流程，如下所示。

```
$ git rebase --interactive 0f187d8
pick 77c00e2 Adding an Easter egg of bad jokes.
pick eed5023 Joke: What goes 'ha ha bonk'?
pick 50605a1 Correcting joke about horses and baths.
pick d1dc647 Revert "Adding office hours reminder."

# Rebase 0f187d8..d1dc647 onto 0f187d8
#
# Commands:
#  p, pick = use commit
#  r, reword = use commit, but edit the commit message
#  e, edit = use commit, but stop for amending
#  s, squash = use commit, but meld into previous commit
#  f, fixup = like "squash", but discard this commit's log message
#  x, exec = run command (the rest of the line) using shell
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

提交列表被倒序排列，最旧的提交现在位于列表的顶端。编辑这列表，将第二个 squash 单词替换为 pick。在我的例子中，编辑后的列表将如下所示。

```
pick 77c00e2 Adding an Easter egg of bad jokes.
squash eed5023 Joke: What goes 'ha ha bonk'?
squash 50605a1 Correcting joke about horses and baths.
pick d1dc647 Revert "Adding office hours reminder."
```

保存并退出你的编辑器以继续。提交消息的编辑器将会在新窗口中打开。你现在需要编写新的提交消息，表示所有你合并的提交。你可以使用当前的说明作为起点，如下所示。

```
# 这是三个提交的组合
# 第一个提交的消息如下所示
Adding an Easter egg of bad jokes.

You should add your bad jokes too.

# 第二个提交的消息如下所示

Joke: What goes 'ha ha bonk'?

# 第三个提交的消息如下所示

Correcting joke about horses and baths.

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# Date:      Wed Sep 10 06:12:01 2014 -0400
#
# rebase in progress; onto 0f187d8
# You are currently editing a commit while rebasing branch 'practice_rebasing'
on '0f187d8'.
#
# Changes to be committed:
#       new file:   badjokes.md
#
```

在我的例子中，更新后的提交消息应该如下所示。

```
Adding an Easter egg of bad jokes.

- New Joke: What goes 'ha ha bonk'?
```

你不需要移除以 # 开头的行。我这么做只是为了看上去更容易阅读一些。

当你对新的提交消息感到满意时，保存并退出编辑器以继续，如下所示。

```
[detached HEAD 1c10178] Adding an Easter egg of bad jokes.
 Date: Wed Sep 10 06:12:01 2014 -0400
 1 file changed, 7 insertions(+)
 create mode 100644 badjokes.md
Successfully rebased and updated refs/heads/practice_rebasing.
```

变基过程现已完成。被你修订后的日志将会如下所示。

```
$ git log --oneline

ef4409f Revert "Adding office hours reminder."
1c10178 Adding an Easter egg of bad jokes.
0f187d8 Added information about additional people to be thanked.
c546720 Adding office hours reminder.
3184b5d Switching back to BADCamp version of the deck.
```

在第二个例子中，我们把单独提交中做出的变更拆分成两个提交。如果你在一个文件中添加了多条修改，并在一个单独的提交中提交了这些变更，但它们事实上应该被保存成两个分开的提交。

和之前一样，将一个提交拆分成多个提交。这次在给出的一列选项中，将其中一个提交前的 `pick` 改为 `edit`。当你保存并关闭编辑器时，Git 会给你修补提交的选项（你知道该怎么做！），然后继续变基过程。

```
Stopped at 0f187d831260b8e93d37bad11be1f41aaeca835e... Added information
about additional people to be thanked.
You can amend the commit now, with

        git commit --amend

Once you are satisfied with your changes, run

        git rebase --continue
```

此时，你处于分离式 `HEAD` 状态下（你已经遇到过这种情况了！没事的！），但文件都被提交了。你需要重置工作目录，让你能够在未提交的文件上工作。你是否记得我们之前用来完成这一步的命令？就是 `reset`! 与其选择一个特定提交，你也可以使用“前一个提交”的缩写，也就是 `HEAD~1`，如下所示。

```
$ git reset HEAD~1

Unstaged changes after reset:
M       README.md
```

现在，你的工作目录中包含未提交的文件，需要在继续变基前被添加。

此时，你可以添加 `--patch` 参数来交互式地暂存你的文件，你可以将一个文件中的修改拆分到两个（或多个）提交。你需要将一个补丁片段添加到暂存区，提交变更，然后将一个新的补丁片段加入暂存区，如下所示。

```
$ git add --patch README.md
```

Git 将会询问你是否想要暂存文件中的每一个补丁片段，如下所示。

```
diff --git a/README.md b/README.md
index 291915b..2eceb48 100644
--- a/README.md
+++ b/README.md
@@ -49,3 +49,5 @@ Emma is grateful for the support she received while employed at
 Drupalize.Me (Lullabot) for the development of this material.
 The first version of the reveal.js slides for this work were posted at
```

```
 [workflow-git-workshop(https://github.com/DrupalizeMe/workflow-git-workshop).
+
+Emma is also grateful to you for watching her git tutorials!
Stage this hunk [y,n,q,a,d,/,e,?]?
```

如果你想要加入这个补丁片段，选择 y，否则选择 n。如果这个补丁片段过大，你只希望加入其中部分片段，选择 s（这个选项在补丁片段过小时不会出现）。浏览文件中的每个修改，选择合适的选项。当你到达变更列表的底部时，你将会回到命令行。使用 git status 命令，假设有多个补丁片段需要修改，你将会看到你的文件准备好被提交且没有被暂存，如下所示。

```
$ git status
rebase in progress; onto bd5c178
You are currently splitting a commit while rebasing branch 'practice_rebasing'
on 'bd5c178'.
  (Once your working directory is clean, run "git rebase --continue")

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   README.md

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   README.md
```

提交你暂存的变更以继续，如下所示。

```
$ git commit
```

如果剩下的变更可以包含在同一个提交中，你可以忽略 --patch 参数，添加文件并将文件提交至仓库，如下所示。

```
$ git add README.md
$ git commit
```

提交了所有变更后，你准备继续变基。似乎没有任何提示，但如果你去检查状态，Git 将会提醒你变基还没有结束，如下所示。

```
$ git status

rebase in progress; onto bd5c178
You are currently editing a commit while rebasing branch 'practice_rebasing'
on 'bd5c178'.
  (use "git commit --amend" to amend the current commit)
  (use "git rebase --continue" once you are satisfied with your changes)

nothing to commit, working directory clean
```

为了完成变基，使用 Git 在状态消息中给出的命令，如下所示。

```
$ git rebase --continue
```

```
Successfully rebased and updated refs/heads/practice_rebasing.
```

终于搞定了！中间有很多步骤，但你之前已经尝试过了这些概念，这次它们被串在了一起。干得漂亮！

如果你完成了本节中的所有例子，你现在应该能够修补提交，使用交互式变基更改分支的历史记录。

6.5.4　撤销分支合并

当你在合并分支时，你可能出现差错。或许你在执行合并时签出了错误的分支，或许你本应该使用 `--no-ff` 参数来执行合并却忘了这么做。只要你还没有发布这个分支，“撤销合并”（unmerge）你的分支将会非常容易。

> **没有所谓的“撤销合并”**
>
> “不可思议”，他喊道。“我不觉得这个词是你说的这个意思”，其他人补充道。类比《公主新娘》，这是真的，Git 中没有所谓的六指人，也没有一种方法可以“撤销合并”什么分支。但是，你可以将分支顶端重置为你使用 `merge` 命令前的那一个提交，从而反转合并产生的影响。希望你不会经常遇到这种情况，因为这个过程可能会耗费你大量的时间。

理想情况下，在错误地合并一个分支后，你将会立即意识到这个错误。这是反转最常见的场景。Git 知道有一些命令相对更加危险，因此它在执行这些危险的操作前保存了一个指向最新提交的指针。Git 认为合并是危险的，因此你可以通过运行 `reset` 轻而易举地取消一个合并，并将你的分支顶端重新指向合并发生前的那个提交，如下所示。

```
$ git reset --merge ORIG_HEAD
```

如果你没有立即意识到你的错误，你需要在继续前问自己几个问题。图 6-6 总结了你在选择正确的命令来撤销合并工作前，需要进行的思考。

你需要仔细思考在继续前你想要保留什么工作，以及想要舍弃什么操作。如果你已经删除了一个分支，你或许希望在另一个分支中建立一份这些提交的备份。这样你就不需要研究引用日志来找到丢失的提交。

假设你正在 *master* 分支上工作，你希望创建一个名为 *preservation_branch* 的备份分支，如下所示。

```
$ git checkout master

$ git checkout -b preservation_branch
```

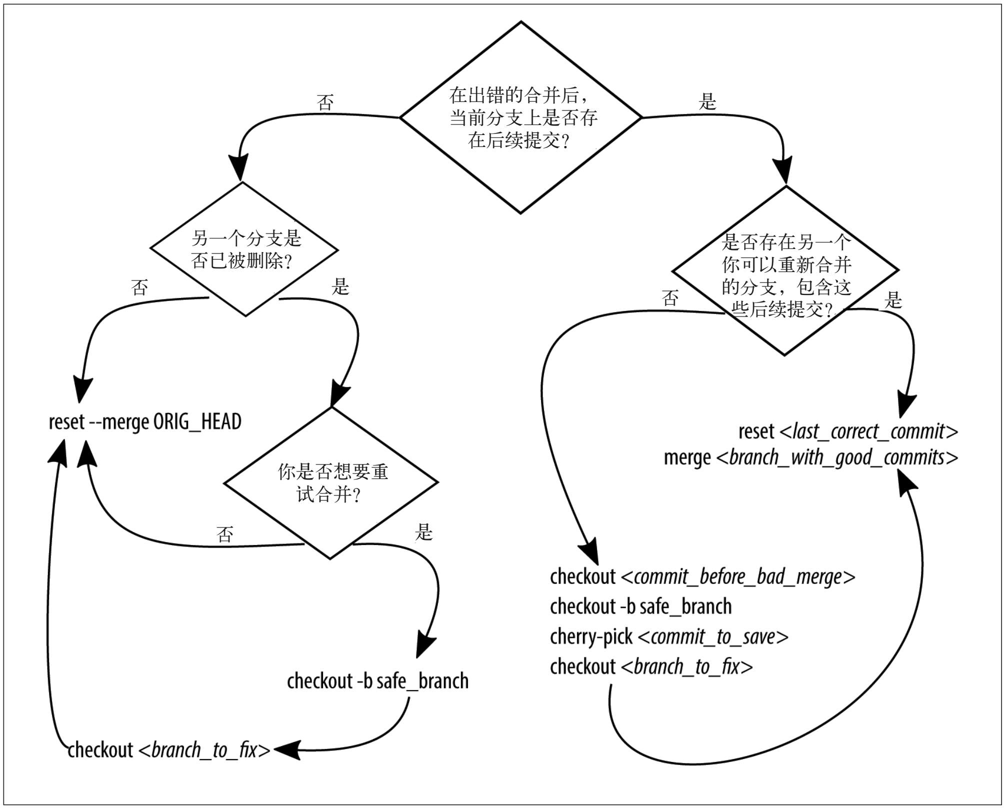


图 6-6：在取消合并分支前，思考如何找回丢失的提交

现在，你的这个分支既包含想要的提交，也包含不想要的提交，你可以通过移除不想要的提交来进入下一步。我们假设你没有其他想要保存在这个分支上的提交，如下所示。

```
$ git checkout master
$ git reset --merge ORIG_HEAD
```

如果你确实希望保存一些提交，现在就可以将这些提交从刚创建的备份分支中 cherry-pick（拣选）过来，如下所示。

```
$ git cherry-pick commit_to_restore
```

只有当你意识到要撤销合并一个出错的分支时，使用 ORIG_HEAD 参照点的做法才能生效。如果你进行了其他工作，有可能 Git 已经建立了一个新的 ORIG_HEAD。在这种情况下，你需要选择想要回退到的提交 ID，如下所示。

```
$ git reset last_correct_commit
```

正如图 6-6 所示，还有一些其他需要撤销合并分支的场景。慢慢了解，记住引用日志跟踪了你的所有操作，即使你丢失了什么东西，你总是可以回到过去，签出一个特定的提交，然后思考怎么做才不会丢失任何工作。

6.6 撤销共享历史记录

本章之前的部分一直集中于改变仓库中未发布的历史记录。在你开始发布你的工作之后，你难免有时候会不小心发布需要修补的工作。有很多原因可以解释这一点，可能是因为来自客户的新需求、因为你发现了 bug 或是因为其他人发现了 bug。如果你要和别人共享你的修改，没有什么可感到羞愧的，你完全不必掩饰自己的学习过程！不过，有时你应该清理一下已经共享的提交历史记录。例如，过多的次要修改会使得 `bisect` 这样的调试工具性能下降，而且干净的提交历史记录更加易于阅读。修改共享历史记录最好的方式是完全不要修改。与其说是“回滚”到一个过去的可用状态，你应该将这个操作视为“前滚”到一个未来的可用状态。你可以通过添加新的提交来完成这个操作，或者使用 `revert` 命令。在本节中，你将会学到如何修补共享的历史记录，而不使你的队友感到困惑。

6.6.1 还原之前的提交

如果你在过去有一个错误的提交，你完全可以使用 `revert`（还原）命令，用一个新的提交恰好抵消之前的提交。如果从物理学的角度讲，`revert` 有点像是降噪耳机。这个命令发出和背景噪声完全相反的声音，事实上你听到的就是一片安静。

当你使用 `revert` 命令时，你将会注意到你的项目历史记录没有被改变。提交没有被移除，而分支顶端出现了一个新的提交。例如，如果你想要还原新增的三个空行和删除的一行，还原操作将会移除这三行并加回被删除的一行。

例如，你的分支历史记录如下所示。

```
50605a1 Correcting joke about horses and baths.
eed5023 Joke: What goes 'ha ha bonk'?
77c00e2 Adding an Easter egg of bad jokes.
0f187d8 Added information about additional people to be thanked.
c546720 Adding office hours reminder.
3184b5d Switching back to BADCamp version of the deck.
bd5c178 Added feedback request; formatting updates to pro-con lists
```

你决定要移除关于办公时间提醒的提交，因为此消息只在过去有意义。此消息被添加在 `c546720` 位置，如下所示。

```
$ git revert c546720
```

提交消息的编辑器将会打开。屏幕上显示了默认的提交消息，你可以保存并退出以继续下一步，如下所示。

```
[master d1dc647] Revert "Adding office hours reminder."
 1 file changed, 2 deletions(-)
```

现在，你记录的历史记录中包含了一个新的提交，用于撤销添加至 `c546720` 中的更改，如下所示。

```
d1dc647 Revert "Adding office hours reminder."
50605a1 Correcting joke about horses and baths.
eed5023 Joke: What goes 'ha ha bonk'?
```

```
77c00e2 Adding an Easter egg of bad jokes.
0f187d8 Added information about additional people to be thanked.
c546720 Adding office hours reminder.
3184b5d Switching back to BADCamp version of the deck.
```

对每个想要还原的提交执行相同的操作。

如果你完成了本节中的所有练习，你现在应该已经可以还原之前已完成的提交中的变更。

6.6.2 撤销共享分支的合并

在本章之前的部分中，你已经学会了通过 `reset` 命令撤销两个分支的合并。这个命令删除了分支历史记录上的提交。因此，Git 再次遇到这些提交时，会把它们视为*新的*提交。当你将一个（过时的）分支并入主仓库时，你就会遇到这种情况。

为了决定使用哪个命令，你需要首先确定你要处理的是哪种合并。图 6-7 将快进合并与真正的合并进行了比较。快进合并基于源分支（即将被并入的分支）上的提交，而真正的合并显示为图上的凸起，在合并发生处引入了一个新的提交。

使用 `log` 命令，查看错误的分支被并入了哪个节点（例 6-8）。如果存在合并提交，那你就走运了！如果没有合并提交，你需要完成更多的工作来撤销分支合并。

例 6-8　提交历史记录中的图形化日志会告诉你是否是真正的合并

```
$ git log --oneline --graph

*   4f2eaa4 Merge branch 'ch07' into drafts
|\
| * c10fbdd CH07: snapshot after editing draft in LibreOffice
| * 9716e7b CH07: snapshot before LibreOffice editing
| * 8373ad7 App01: moving version check to the appendix from CH07
| * d602e51 CH7: Stub file added with notes copied from video recording lessons.
* | 1ae7de0 CH08: Incorrect heading formatting was creating new chapter
* | 7907650 CH08: Draft chapter. Based on ALA article.
* | ad6c422 CH8: Stub file added with notes copied from video recording lessons.
```

如果你希望查看某个单独的提交是否是一次真正的合并，你可以使用 `show` 命令。这个命令将会列出所有合并的分支的 SHA1，如下所示。

```
$ git show 90249389

commit 902493896b794d7bc6b19a1130240302efb1757f
Merge: 54a4fdf c077a62
Author: Joe Shindelar <redacted@gmail.com>
Date:   Mon Jan 26 18:30:55 2015 -0700

    Merge branch 'dev' into qa
```

谢谢你的建议，Joe！

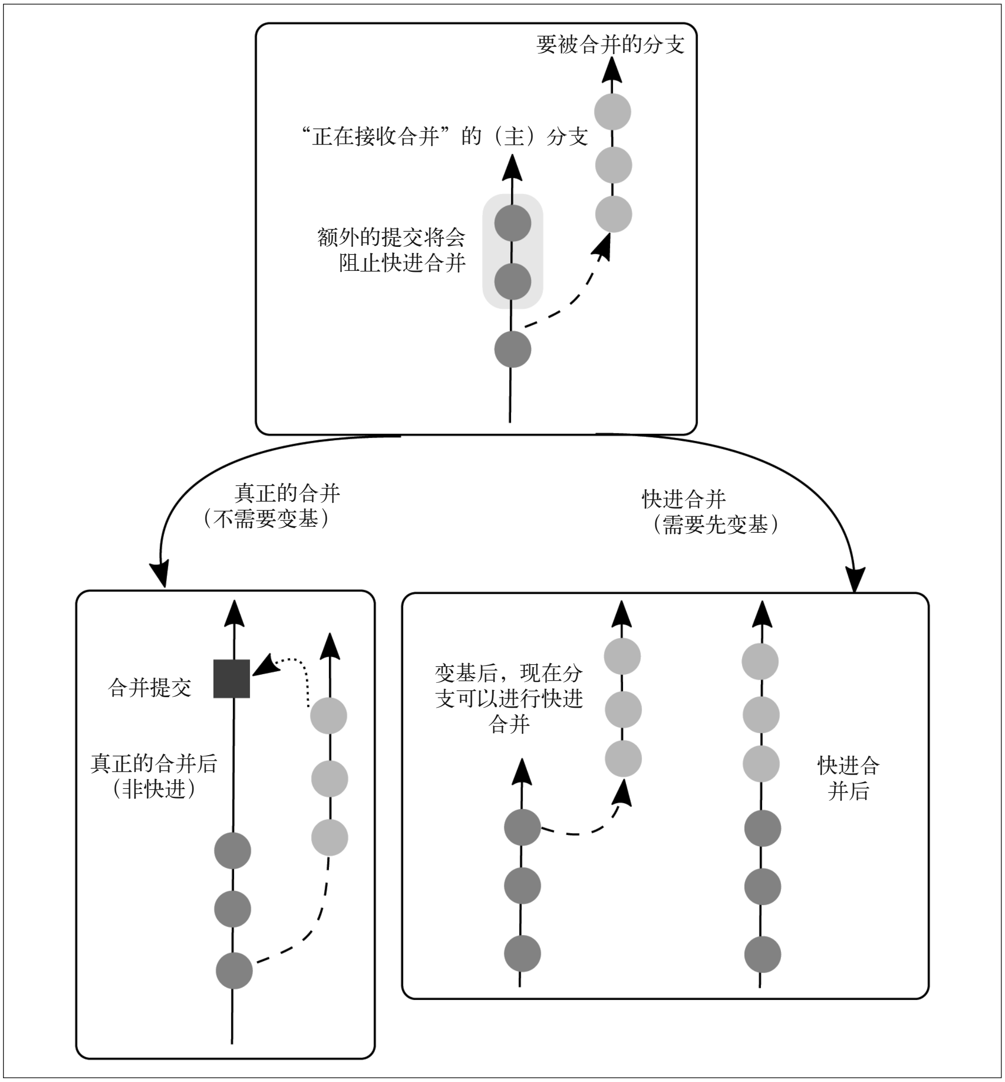


图 6-7：如图所见，快进合并丢失了分支原本的面貌，真正的合并保留了这个面貌

保持一致，方便搜索

合并提交默认的提交消息是“Merge branch *incoming* into *current*”（“将某个分支并入当前分支”），当你在查看 `log` 命令的输出时你将更容易找到合并提交。你的团队可能选择使用不同的提交消息模板，但是，你可以添加可选的 `--merges` 和 `--no-merges` 参数来过滤输出的历史记录。

一旦你找到了一个合并提交，你就可以选择一组合适的命令。图 6-8 将这些选项总结成一张流程图。

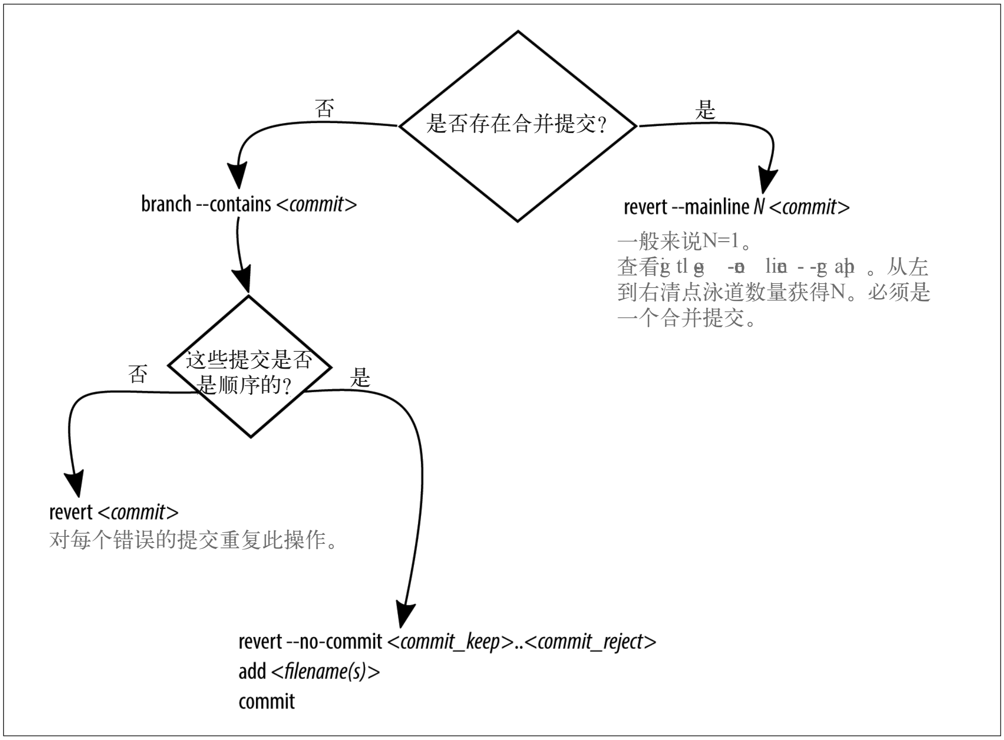


图 6-8：根据分支的合并方式，你将会使用不同的命令来撤销共享分支的合并

如果分支被真正地合并了，而不是快进合并，那么撤销的流程如下：使用 `revert` 命令反转合并提交的作用（例 6-9）。这个命令接受一个额外的参数 `--mainline`。这个参数告诉 Git 在撤销这次合并时应该保留哪个分支。仔细观察图形化的日志，数一数从左到右的泳道。第一个泳道为 1。你几乎总是希望留在最左边的泳道，因此因此这个数字几乎也总是 1。

例 6-9　反转一个合并提交

```
$ git checkout branch_to_clean_up
$ git log --graph --oneline
$ git revert --mainline 1 4f2eaa4
```

提交消息的编辑器将会打开。默认的提交消息指出你刚执行了一次还原操作，并提供了你撤销的那个提交的提交消息（例 6-10）。我一般偷懒保留这个说明，但是，这样做也有好处，当我搜索历史记录时找到反转合并的那个提交将会非常容易。

例 6-10：还原合并提交的提交消息示意

```
Revert "Merge branch 'video-lessons' into integration_test"

This reverts commit 0075f7eda67326f174623eca9ec09fd54d7f4b74, reversing
changes made to 0f187d831260b8e93d37bad11be1f41aaeca835e.

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
```

```
# On branch master
# Your branch and 'origin/master' have diverged,
# and have 23 and 2 different commits each, respectively.
#   (use "git pull" to merge the remote branch into yours)
#
# Changes to be committed:
#       deleted:    lessons/01-intro/README.md
#       deleted:    lessons/02-getting-started/README.md
#       deleted:    lessons/03-clone-remote/README.md
#       deleted:    lessons/04-config/README.md
(etc)
#
```

有时，当你运行 revert 时会遇到冲突。不必慌张。只要把它当作是平常的合并冲突就好，遵循 Git 在屏幕上输出的指示，如下所示。

```
$ git revert --mainline 1 a1173fd

error: could not revert a1173fd... Merge branch 'unmerging'
hint: after resolving the conflicts, mark the corrected paths
hint: with 'git add <paths>' or 'git rm <paths>'
hint: and commit the result with 'git commit'
Resolved 'README.md' using previous resolution.
```

出了点问题，请检查状态消息，看看我们需要审查哪些文件，如下所示。

```
$ git status

On branch master
Your branch and 'origin/master' have diverged,
and have 20 and 2 different commits each, respectively.
  (use "git pull" to merge the remote branch into yours)
You are currently reverting commit a1173fd.
  (fix conflicts and run "git revert --continue")
  (use "git revert --abort" to cancel the revert operation)

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        deleted:    badjokes.md
        modified:   slides/slides/session-oscon.html

Unmerged paths:
  (use "git reset HEAD <file>..." to unstage)
  (use "git add <file>..." to mark resolution)

        both modified:   README.md
```

状态消息中提到仓库与 `origin` 不同步，这与我们的问题无关。跳过这条，继续阅读。第一个有用的信息从这里开始：`You are currently reverting`（您在执行反转操作）。你会看到如何继续的选项以及如何中断这个过程的选项。不要放弃！继续读下去。接下来几行看上去像是平常混杂的工作目录，其中一些文件被暂存了，而另一些没有。如果你只是在编辑你自己的文件，你会知道如何处理。首先，将你的修改添加到暂存区，然后提交，如下所示。

```
$ git add README.md
$ git commit -m "Reversing the merge commit a1173fd."

[master 291dabe] Reversing the merge commit a1173fd.
 2 files changed, 2 insertions(+), 7 deletions(-)
 delete mode 100644 badjokes.md
```

如果没有合并提交，你需要一个个处理每个你想要撤销的提交。这个过程将会尤其令人沮丧，因为快进合并在图形化日志中没有提供任何可视化的线索，告诉你那个烦人的分支中有哪些提交。（在第一次撤销拣选错误的合并后，你将会理解合并分支时看到的 `--no-ff` 策略的逻辑。）

> **和你的团队讨论应该怎么做**
>
> 在一个个撤销拣选这些提交前，你可能想看看团队中有谁有尚未发布且未被污染的分支，让他们共享这个分支。有时，小心地使用 `push --force` 可以更容易地改变历史记录。

你需要做的第一件事是猜测出错的提交在哪。如果你完全确定哪里出错了，那么可以通过使用 `--contains` 参数的 `branch` 命令来列出包含某个提交的所有分支，如下所示。

```
$ git branch --contains commit
```

假设被并入的那个分支没有被删除，那么你应该能够使用这些信息来判断你要撤销合并哪个分支，以及哪些提交被添加到你或许想要移除的分支。但是，记住，这些提交同时出现在了两个分支中，因此你无法通过比较来找出哪些提交是不同的。如果你已经知道你想要找的提交，你不必完成这一步。

如果你想要还原的提交是连续的，你太走运了！ `revert` 命令可以接受单个提交，或是一系列提交。但是，记住，`revert` 将会为每个反转的提交创建一个新的提交。在你的提交历史记录中，这些提交看上去可能非常讨厌，因此与其单独反转每个提交，你可以选择将这些提交组合成一次反转，最后再保存提交消息，如下所示。

```
$ git revert --no-commit last_commit_to_keep..newest_commit_to_reject
```

在运行这个命令后，你将会得到一个混乱的工作目录，所有文件都被还原。审查这些变更。然后，完成还原过程，如下所示。

```
$ git revert --continue
```

审查提交消息，补充必要的信息来使之更清晰。在默认情况下，这个说明将会以“Revert”开头，接着是一串带引号的文字，描述你正在反转的最新提交中的内容。通常这样就足够了，但如果原来的说明不够好，你可能会希望让描述更加清晰一些。

如果提交不连续，你需要一次一个地还原这些恼人的提交。给我发条推文，提到 @emmajanehw，我会给你一些安慰并给你鼓掌。

```
$ git revert commit
```

除非特定的工作流要求，撤销合并一个已合并的分支不是 Git 希望看到的事。你的团队可

能永远用不到撤销合并一个分支。显然，在我的个人项目中有时会遇到错误的合并，作为独立开发者，我已经习惯了暗自悲伤一会，然后将烦恼抛在脑后。有时历史记录并没有那么重要，而有些时候确实很重要。有了经验和教训，你对本该使用什么命令了然于心。

6.7 真正移除历史记录

在本章中，你已经学到了更新仓库历史记录，尤其是如何在你认为工作丢失时获取信息。有时，或许你真的想要丢掉部分历史记录——例如，如果你不小心提交了一份很大的数据文件或者一份包含密码的配置文件。希望你永远用不到这一节中的内容，但以防万一你的“朋友”需要帮助呢，我加入了这些指南。你懂得，不怕一万，就怕万一。

> **发布后的历史记录是公开的**
>
> 如果你将一些内容发布到一个公开的远程仓库，你应该做好这样的准备，其他人可能会上去克隆一份仓库的副本，并且访问到你不打算公开的秘密。立刻更新在这个仓库中公开过的密码和 API 密钥。

如果你需要在已发布的分支上清理一些东西，你应该在第一时间通知你的队友。你应该让他们知道你将要进行一些清理，并且将向仓库“强制推送”一份新的历史记录。开发者需要重新评估他们的本地仓库，并决定他们所处的状态。让每个开发者都找一找冲突的文件，看看他们的仓库是否被污染了，操作步骤如下所示。

- 如果你尝试移除的文件不在他们的本地仓库中，他们不会被你的清理所影响。
- 如果他们的仓库中的每个分支都包含这份文件，那么他们的仓库就被污染了。但是，如果他们在这份文件被引入前没有开始任何工作，他们不会被你的清理所影响。这一点对于不是本地开发者的 QA 管理者来说也成立。在这种情况下，让他们移除自己本地的仓库副本，并在清理完成后重新克隆仓库。
- 如果他们的仓库被污染了，并且包含被污染历史记录的分支上包含他们本地的工作，他们需要通过变基更新这些分支。如果他们使用 `merge` 来更新他们的分支，他们会再次将有问题的文件引入仓库，而你的工作也将前功尽弃。对于不熟悉变基的人来说这会有些可怕，因此你应该建议他们将包含需要保存的工作的分支推送上去，这样你可以帮他们清理这些分支。（在清理完成后让他们克隆一份新的仓库。）

在你进行清理时，你的同事们可以做些别的事（https://xkcd.com/303/）。

当团队中每个人都收到通知，并且你准备好了一份计划，应对清理其他每个人的仓库在清理之前、当中及之后将会发生的事，你可以继续下去了。

为了完成这个过程，你可以使用 `filter-branch`。这个命令允许你重写分支历史记录和标签。Git 文档中提供的例子十分有趣，值得阅读。例如，你可以使用这个命令来永久移除某个作者提交的任何代码。我不认为我会不评估影响就移除别人的代码，但有趣的是这个命令确实可以这么使用。（不过，或许你恰好知道应该如何利用这个命令？）

假设你想要移除的文件是 SECRET.md，命令如下所示（这是一个很长的命令，\ 符号允许你换到第二行）。

```
$ git filter-branch --index-filter \
  'git rm --cache --ignore-unmatch SECRET.md' HEAD
```

在文件从仓库中完全移除后，将它加入你的 .gitignore 文件，这样它们就不会被再次意外地添加了。使用 .gitignore 的指南可以在附录 C 中找到。

与本章中的其他方法不同，现在我们希望永久地从仓库中移除一些冲突的内容。首先，这些提交仍然可以通过 reflog 命令找到。当你确定不再需要这些提交之后，可以通过清除本地历史记录并执行一次垃圾回收（gc），将这些提交从你的系统中丢弃，如下所示。

```
$ git reflog expire --expire=now --all
$ git gc --prune=now
```

你的仓库现已清理完成，你可以将新版本推送到你的远程仓库，如下所示。

```
$ git push origin --force --all --tags
```

一旦项目新版本的历史记录可以从共享仓库中访问到，你可以让你的同事们更新他们的工作。根据你之前的对话，他们会通过以下方法来并入你处理过的变更。

- 再次从零克隆一份仓库。这个方法适合不使用变基并对它感到畏惧的团队。
- 使用 rebase 更新他们的分支。这个方法适合熟悉变基的团队，因为这比重新克隆一份仓库要快，并且能够保留他们本地的工作，如下所示。

```
$ git pull --rebase=preserve
```

GitHub（https://help.github.com/articles/remove-sensitive-data/） 和 Bitbucket（https://confluence.atlassian.com/display/BITBUCKET/Maintaining+a+Git+Repository）都提供了文章来说明如何清理存放在它们网站上的仓库。由于这两个网站涉及的场景略有不同，两篇文章都值得一读。

现在，你已经了解了 Git 自身清理仓库的方式，查看这个单独的包，BFG Repo Cleaner（https://rtyley.github.io/bfg-repo-cleaner/）。它提供了和 filter-branch 相同的结果，但使用起来更加快速，而且在安装之后更加易于使用。如果你对 filter-branch 所需要的清理时长不满意，你绝对应该试一试 BFG。

6.8 命令指南

表 6-2 列出了本章提到的所有命令。

表6-2：撤销工作需要的git命令

命令	用途
git checkout -b branch	创建一个名为 *branch* 的分支
git add *filename*(s)	暂存文件，准备提交至仓库
git commit	将暂存的变更保存至仓库
git checkout *branch*	切换到指定分支
git merge *branch*	将 *branch* 中的提交并入当前分支

（续）

命令	用途
git branch --delete	移除本地分支
git branch -D	移除不包含并入其他分支的提交的本地分支
git clone *URL*	下载一份远程仓库的副本
git log	查看项目历史记录
git reflog	查看分支的详细历史记录
git checkout *commit*	切换到另一个本地分支
git cherry-pick *commit*	将提交从一个分支复制到另一个分支
git reset --merge ORIG_HEAD	移除当前分支中所有在最近一次合并中引入的提交
git checkout --*filename*	还原已变更但尚未提交的文件
git reset HEAD *filename*	从暂存区移除提出的文件修改
git reset --hard HEAD	将所有已变更的文件还原到之前保存的状态
git reset *commit*	取消暂存在这个提交之前的所有提交中的变更
git rebase --interactive *commit*	编辑，或压缩提交后的所有提交
git rebase --continue	在解决合并冲突后，继续变基过程
git revert *commit*	取消应用指定提交中的变更，创建一个共享友好的历史记录还原
git log --oneline --graph	显示分支的图形化历史记录
git revert --mainline *1 commit*	反转一个合并提交
git branch --contains *commit*	列出所有包含指定提交对象的分支
git revert --no-commit *last_commit_to_keep..newest_commit_to_reject*	使用一个提交反转一组提交，而不是为每个撤销的提交都创建一个对象
git filter-branch	从仓库中永久移除文件
git reflog expire	忽略详细历史记录，仅使用存储的提交消息
git gc --prune=now	运行垃圾回收器并确保所有未提交的变更从本地内存中移除

6.9 小结

在本章中，你学到了如何与 Git 仓库历史记录打交道。我们介绍了 Git 中一些通常被新手视为“高级”的命令的常见应用场景。通过画图总结你的仓库状态和你想要做出的变动，你可以有效地选择每种场景下要允许的正确 Git 命令。你学会了如何使用 Git 的三个“R”。

- Reset（重置）
  将分支顶端移至一个之前的提交。这个命令不要求提交消息，如果没有使用 `--hard` 参数可能返回一个混杂的仓库。

- Rebase（变基）
  允许你改变分支历史记录中提交的存放方式。通常用于将多个提交压缩成一个提交，来清理分支，并且保持与另一个分支处于最新状态。

- Revert（还原）
  还原共享分支上一个特定提交中做出的变更。这个命令与提交搭配使用，返回一个干净的工作目录。

在下一章中，你将会开始整合本地仓库中的工作与团队中其他人的工作。

第 7 章

多人团队

前几次与其他人合作完成项目的经历将会形成你使用版本控制的方式。如果你的合作者十分耐心并且富有同理心，你更有可能会自信地使用版本控制。富有同理心的队友会在文档中记下他们希望你使用的流程，并且耐心回答你的问题（在必要时更新文档）。如果你负责启动一个项目，可以想想 Jerry Maguire 对他的明星球员所说的“我来帮你”。作为项目负责人，这应该是你的口头禅。找到症结并解决它们。在你希望保持一致的地方，提供详细的指示、模板和自动化的脚本。当有些地方没有达到你的期望时，将它看作是你应该解决的流程问题。

在本章中，我们将本书之前学到的内容推向了高潮。在第一部分中，你学到了关于设置项目时的不同考虑事项。现在，你将会学习如何落实这些决定。在第 5 章和第 6 章中，你学到了开发者每天运行的命令。在本章中，你将会学习如何建立远程项目连接，并将你的工作与他人共享。

完成本章学习之后，你将具备以下技能。

- 在代码托管系统上设置一个新的项目
- 使用 `clone` 下载一个远程仓库
- 使用 `push` 将你的变更上传到一个仓库
- 使用 `fetch` 刷新远程仓库中的可用分支列表
- 使用 `pull` 合并远程仓库中的变更
- 解释使用 `pull`、`rebase` 和 `merge` 更新分支带来的影响

本章尽可能提供了帮助新来开发者上手的模板。贡献有用的工作越容易，人们就越享受为你的项目工作。即使这只是一份工作，我们的生活也没有理由不可以有更多乐趣。

喜欢通过视频教程学习的读者可以参考本书附带的系列视频，Collaborating with Git（http://

shop.oreilly.com/product/0636920034872.do）。

7.1 设置项目

项目的背景将在很大程度上决定如何设置仓库。为了保证隐私，我们会建立一个绝密的内部代码库；为了体现透明并鼓励参与的风格，我们会建立一个免费的开源代码库。 一旦项目建立，开发人员每一天中使用的命令可能会非常相似。

本节介绍了在代码托管系统中创建一个新项目的基本流程。GitHub、Bitbucket 和 GitLab 中的细节将在第三部分中（分别在第 10 章、第 11 章和第 12 章中）介绍。

7.1.1 创建新项目

为了与你的团队共享工作，你需要在选择的代码托管系统中建立一个新的项目。如今，大多数代码托管系统远远不只是一个能够储存共享仓库的地方。它们还提供了工单系统、基本的工作流优化、项目文档仓库和更多功能！我参与过的社区和团队通常使用以下三个服务之一：GitHub（通常用于开源项目）、Bitbucket（通常用于内部团队和想要免费私有项目托管的小团队）和 GitLab（通常用于因为安全原因需要在内部托管代码的中型公司）。

不论你选择哪个系统，设置项目的基本流程都是一样的。你需要先问自己一个问题：你要用什么账户来创建这个仓库？在基于 Web 的系统中，项目 URL 的基本格式如下：https://<hosting-url.com>/<project-owner's-name>/<projectname>。如果这个项目确实是你的（例如，个人博客的仓库），那么你可以在 URL 中包含你的用户名。然而，如果这个项目属于一个开发者组织，用组织的名称来命名项目所有者会更合适。最后，如果项目从属于不止一个组织，如对于一个开源软件项目而言，使用软件项目来命名项目所有者最为合适。

你选择的决策可能还会影响对项目拥有直接写入权限的人，他可能独立于你所使用的代码托管系统。例如，如果你选择使用你个人的名字来启动一个项目，你可能不希望随便谁都可以不经你的审查写入项目，尤其是在公开项目中，其他人评价工作内容时可能会假设这些工作都是你的。

名字有什么意义？

这些年本书的支持仓库出现在很多不同的地方，包括我的个人账户、一个团队账户和三个不同的代码托管系统（总共有六个不同的仓库需要维护）中。尽管这些工作是由我开发的，但想要分发到绑定的哪个 URL 渐渐成为了问题。如果我希望其他人将这个仓库视为他们的（例如在一些人们无法找到我的某套抽象的学习资料中），我或许会使用项目 URL。但当我希望人们知道我是作者，因为这也是自我推广的好机会，我或许会把我的个人 URL 给他们。很有可能我过度解读了名字的意义，但你至少应该考虑一下各种东西的命名。

在阅读本书时，你或许是团队中的一员（即使是不能再小的单人团队！），作为项目的所

有者，你希望为你的公司、组织或团队选择一个名称，如果你正在进行一个开源项目，你还要为项目选择一个名称。幸运的是，你可以轻易地将代码库移到另一个名称或者甚至是另一个代码托管平台，因此一开始就想好正确的名称并不是至关重要的。然而，将项目的元数据从一个账户迁移到另一个账户会更加困难。元数据可能包括项目中工单的历史记录以及仓库外存放的文档。

选好项目所有者，然后在此账户下建立一个新的空项目。不要担心文件还没有上传。

7.1.2 建立权限管理

你需要为你的项目设置两种权限：谁能够看到这个项目（“读”），以及谁可以提交到这个项目（“写”），对此我们在第 2 章中已进行过详细的讨论。如果你的团队是一个超级透明的团队，项目应该对全世界可见。否则请创建一个私有项目。

免费服务的代价

一些代码托管服务会对私有项目收取少量费用，而另一些则免费提供服务。如果你的代码和历史记录很重要，请考虑付费托管。你可以选择付出时间的代价搭建内部的代码托管，也可以选择以付出少量月费为代价使用一个第三方服务。付费的好处是托管公司更有可能把你当作一个客户，通过付费弥补他们的支出，你能够更好地为他们的业务生存提供帮助。当然，如果你无法负担这个费用，你也有很多免费的选项，如果一家公司选择提供的免费服务，你也没必要感到羞愧。做你能做的。

此外，一些托管系统允许你设置分支特有的限制。目前 Bitbucket 和 GitLab 提供这个功能。设置选项分别在第 11 章和第 12 章中介绍。

作为一个分布式版本控制系统，Git 本身在处理仓库中产生的变更请求时表现良好。一般来说，团队项目会有一个称为“项目”（主项目）的单独仓库，和许多包含每个开发者工作的衍生项目。如果你在内部项目中工作，则可以选择让所有人直接在“项目”的仓库中工作；但如果你倾向于维护一个干净的中央仓库，则可以选择让每个开发者在一份派生的“项目”中工作。

项目

在本章中，你将会多次见到“项目”这种说法。我使用这个缩写来指代一个软件项目主要的或者官方的仓库。社区同意将这个仓库用作软件的官方发布。Git 本身没有内部机制来强制规定一个仓库比另一个仓库更重要。只有社区的声明会使一个仓库成为官方仓库。

根据你在第 2 章中做出的关于团队结构的决策，为所有贡献者分配合适的权限，决定谁可以拥有“仓库”的写入权限，其他未认证的开发者的贡献可以通过拉取请求（在一些服务中也称为合并请求）来接受。

7.1.3 上传项目仓库

作为一个分布式版本控制系统，Git 像是一个交际花。它喜欢连接各种仓库，它喜欢分享故事，并一路结交新的朋友。Git 通过储存一个称为*远程*（remote）的连接来联系远方的朋友。本地仓库可以有零个、一个或多个远程连接。通常情况下，Git 仓库只有一个远程连接，即 origin（源）。你可能已经见过了这个术语。分配给远程仓库的这个别名，从中下载或克隆了你的本地副本。你可以使用任何你喜欢的名称来命名你的远程连接。

在你第一次启动一个新项目时，可能还没有现成的代码，也可能有了一些代码。（这显而易见，不是么？）如果你还没有代码，那么可以遵循代码托管系统中的指示，将这个空项目克隆到你本地的开发环境中。然而，如果你在本地已经有了一些代码，那么将会想要上传你已有的代码。要想这么做，你需要建立一个从你的本地仓库到项目托管服务的新连接。

在项目仓库的本地副本中，检查你是否已经设置好一个远程连接，如下所示。

```
$ git remote --verbose
```

如果你从本地启动项目，就不会看到任何远程连接，因此，如果现在什么都没有显示也没关系。如果你为这个仓库设置了一个远程连接，将会看到如下内容。

```
origin  https://github.com:emmajane/gitforteams.git (fetch)
origin  https://github.com:emmajane/gitforteams.git (push)
```

每行都以远程连接（*origin*）的别名开头，后跟远程仓库的原地址。这些行总是会成对出现：每一对的第一行表明你将从哪里获取新的工作（`fetch`），而第二行表明你会将新的工作上传到哪里（`push`）。

项目所有者需要连接到项目的官方副本，如果他们在合并工作前需要经历同行评审的过程，那么可能还需要连接到一个派生的项目（同行评审在第 8 章中介绍）。一旦你开始为某个项目添加多个远程仓库，默认的别名（*origin*）会令人有些困惑。因此，我倾向于根据仓库的不同目的来命名，例如对于你来说有意义的 *official*（官方）和 *personal*（个人）。在我上传工作时，会在这两个选项中选择一个使用。在标准 Git 术语中，我的别名是 *upstram* 和 *origin*，尽管 *origin* 默认被分配给克隆仓库的源，不管你是否拥有写入权限。

> **命名**
>
> 我已经用了很长时间的 Git，但我还是动不动就记错 `git remote show origin` 这个命令。四个单词。记住顺序对我并不难，对吧？但我似乎从来都记不住 `show` 和 `origin` 的顺序。用我自己的名称命名远程仓库之后，我更可能理解这个命令的意义，从而记住正确的顺序。`git remote show official` 对我来说似乎就更好记一些。你可能从来没有遇到过这个问题，但如果你记忆这个命令很费劲，那么或许会希望自定义你的远程名称，将 *origin* 改为你顺口的名称。

要添加一个远程连接，你需要首先知道项目的 URL。URL 的结构通常是 https://<hosting-url.com>/<project-owner’s-name>/<projectname>.git。在新版本的 Git 中，你还可以使用

https 协议，但以前的文档中，第一部分可能被替换为 git@hostingurl. com。一旦你知道了远程仓库的 URL，就可以连接上去（例 7-1）。

例 7-1　添加一个远程仓库连接

```
$ git remote add nickname project-url
```

在连接上远程仓库后，你应该可以在列出远程连接时看到新增的两行。如果你希望使用 Git 的术语，那么应该使用别名 *upstram* 来代表官方项目仓库；如果你使用我的命名约定，那么应该使用 `official`。这个名称永远不会被发布，没有 Git 警察会来纠正你，因此你可以取任何你想要的名称，没有人会知道。[如果你喜欢，甚至可以叫它 *cookies*（曲奇）或 *coffee*（咖啡）。真的无所谓。]

例如，如果我是一个名为 Mounties 的项目中的一员，这个项目由一个名为 Oh, Canada 的机构运作，我的一系列远程连接会如下所示。

```
$ git remote --verbose

official https://github.com:ohcanada/mounties.git (fetch)
official https://github.com:ohcanada/mounties.git (push)
personal https://github.com:emmajane/mounties.git (fetch)
personal https://github.com:emmajane/mounties.git (push)
```

只要你喜欢，就可以很容易地建立许多远程连接。例如，你可能有 *devserver*（开发服务器）、*staging*（暂存）和 *production*（生产）的远程连接，或者你可能会直接登录到那些机器上，拉取主项目仓库中的代码，而不是直接将代码推送到这些地方。

如果你已经在本地仓库中设置好了一个不再需要的远程连接，那么可以轻易地删除它（例 7-2）。

例 7-2　移除一个远程连接

```
$ git remote remove nickname
```

> 你可以轻易地重命名远程连接，甚至为本地仓库中的每个分支设置默认的远程连接。Git 关于这个命令的自带文档清晰易懂。如果你希望更进一步地自定义你的远程列表，那么应该阅读一下这份文档。

在你的项目中创建了远程连接之后，现在你可以将仓库的本地副本上传到远程服务器，如下所示。

```
$ git push nickname branch_name
```

如果你希望与别人共享所有的本地分支，可以修改这个命令，如下所示。

```
$ git push --all nickname
```

一旦你上传了你的工作，请浏览项目页面，确保仓库按照预期上传了。默认情况下，大多数代码托管系统会显示分支 *master*，如果仓库中有超过一个分支，并且你的本地仓库使用了非标准的分支名，那么去看看你的代码托管系统是否允许你为仓库分配默认分支。这个

分支通常是项目最稳定的版本，而试验性的工作会出现在其他分支。当然，每个项目略有不同。你的项目可能将 *master* 分支当作新工作的接合处，而不是软件最稳定的版本。请在你的文档中明确记录这些信息。

要将本地名称上传到远程服务器上的某个新名称下，请使用以下语法。

```
$ git push nickname branch_local:branch_remote
```

例如，如果你希望将你的 *main* 分支上传到远程仓库 *official* 并在远程仓库中将这个分支命名为 *master*，则会使用下面的命令。

```
$ git push official main:master
```

现在，你的本地仓库应该已经被上传到了远程项目仓库，并且使用了你想要的分支名。

7.1.4　在README中记录项目

当你访问项目页面时，将会注意到大多数代码托管系统将会显示 README 文件的内容（如果你的项目中存在 README 文件）。这个文件应该用于向人们概述项目。如果这是一个包含依赖的开发项目，那些项目依赖应该在 README 文件中列出。如果有安装说明，那么也应该在 README 文件中列出（或提供链接，指向更完整的安装指南）。如果你希望人们为项目做出贡献，或者向项目报告 bug，那么这些说明也应在 README 文件中列出。

下面的项目有着优秀的 README 文件，它们清楚地解释了这个仓库是什么，如何使用其中的代码，以及你可以如何对其做出贡献，如下所示。

- Sculpin（https://github.com/sculpin/sculpin）
- Sass（https://github.com/sass/sass）
- Rails（https://github.com/rails/rails）

为你的项目选择一份许可证

世界上没有通行的版权法律。因此，任何不包含明确许可证的项目都被假定为完全受版权保护，不希望被二次使用。坦诚地说，我的不少项目都没有包含许可证。通常这只是因为我还没有想好如何让别人使用我的工作。（在我编写培训材料的圈子里，版权归属的限制比开源许可盛行的代码社区中更加严苛。）一个仓库的许可证通常位于 LICENSE 文件或 LICENSE.txt 文件中。

如果你的本地仓库还没有 README 文件，现在就是添加的好机会！现在，新的项目流行 Markdown 格式的 README 文件，因此将文件重命名为 README.md 来确保文件的格式正确。

在项目上传并且创建这些说明之后，现在是时候开始把贡献者加入到项目中来了。你在本章剩余部分中使用的流程应该以文档的形式加入你的仓库。它会允许开发者拥有一份本地副本，并允许他们更容易地访问到这些信息，而不必参考外部 wiki 页面。

现在，你的项目已经就绪，是时候换个视角，从贡献者的角度来看待事情了。

7.2　设置开发者

当你从开发者的角度审视项目时，并不总是完全清楚应该参与到什么程度。在公开的项目中，开发者可能的参与程度分为以下三个层次。

- 下载项目的压缩包，再也不回到项目页面。这可能出现在真正的派生项目中，下游开发者不打算回去查看代码的演进。它还可以用于被设计为起始点的项目，开发者的目的是分析代码并且大量修改源码。
- 为了保持本地代码最新而克隆项目仓库，但不打算修改项目。这个想法适用于在项目中用到开源库的开发者。这些开发者可能会扩展库，或许对克隆库进行少量修改。但大多数情况下，他们直接使用库的代码，依靠上游开发者的改进和安全更新。
- 为了贡献工作而克隆项目仓库。这个想法适用于开源项目的志愿者和工作人员，软件项目的内部开发者，以及为某个特定项目的构建做出贡献的机构员工。

后两个选择最主要的区别在于，非贡献者一般会选择直接克隆主项目，而贡献者除了项目仓库之外，可能还有一个个人仓库。这些选择的原因在第 2 章中有更详细的介绍。

> **消费者与贡献者**
>
> 具有前瞻性（中级到高级）的开发者总会假设他们会在某个时候贡献回项目，并创建自己中间的远程仓库。然而，大多数新手开发者则尽可能简化自己的工作流程，忽略了创建自己的远程仓库的中间步骤。这也意味着他们将自己视为你的项目的消费者，而不是一个潜在的贡献者。

一旦开发者把自己当作消费者或贡献者（包含主要维护者），就可以选择一种方法来下载你的项目仓库。

7.2.1　消费者

消费者不打算为项目做出贡献。他们不期望拥有代码库的写入权限，在上传自己的更改时也没有预计未来的后果。这类开发者可能会通过以下两种方式来下载你的仓库。

- 以压缩包的形式。
- 以仓库的克隆（直接从“项目”页面）的形式。

压缩包与“项目”没有任何联系，也不包含一直以来的变更历史。另一方面，克隆（clone）维持了项目的连接，并可以通过运行一些 Git 命令更新到最新的版本。克隆一个远程仓库的结构如下所示。

```
$ git clone https://<hosting-url.com>/<project-owner's-name>/<project-name>.git
```

例如，如果你想要为“Git 团队协作”工作坊下载一份项目仓库的副本，可以使用下面的命令。

```
$ git clone https://github.com/gitforteams/gitforteams.git
```

为了更新仓库的本地副本，首先你需要 fetch（拉取）“项目”最新的变更（此处，我们假设你只有一个远程连接），如下所示。

```
$ git fetch --all
```

一旦你拉取完这些变更，在决定更新你的本地副本前，可以比较本地版本与最新版本，看看发生了哪些修改。

首先，列出仓库中所有分支，如下所示。

```
$ git branch --all
```

你将会看到两组分支：你的本地分支和远程跟踪分支。当前被签出的分支将会用 * 标记。我之前自己克隆的项目仓库如下所示。

```
  gh-pages
* master
  video-lessons
  remotes/personal/gh-pages
  remotes/personal/master
  remotes/personal/video-lessons
```

列表显示了三个本地分支和三个连接到 personal 远程仓库的分支。

为了获取每个分支中的更多细节，请使用 --verbose 参数，如下所示。

```
$ git branch --all --verbose
```

输出包括提交说明以及每个分支的状态（与远程仓库相比），如下所示。

```
  gh-pages                          629b54f Resolving merge conflict;  ...
* master                            2db982d Changes to "Undo" graphic: ...
  video-lessons                     7798eb1 [ahead 11] Lesson 00:      ...
  remotes/personal/gh-pages         629b54f Resolving merge conflict;  ...
  remotes/personal/master           2db982d Changes to "Undo" graphic  ...
  remotes/personal/video-lessons    653f875 Lesson 7: Added intro on   ...
```

为了查看添加到仓库的 master 分支上的变更历史记录，你可以使用 log 命令，如下所示。

```
$ git log personal/master
```

为了比较你本地的分支副本和刚下载的更新，你可以添加 --patch 来查看每个提交的更改，或使用 diff 命令查看所有变更的摘要，如下所示。

```
$ git log --patch personal/master
$ git diff master personal/master
```

这个命令会以补丁的格式展示所有变更。寻找被添加（标记为 +）或被删除（标记为 -）的行。如果你倾向于签出整个代码库，可以签出分支顶端，如下所示。

```
$ git checkout personal/master
```

你将会处于分离式 HEAD 状态。签出 *master* 分支以回到其本地副本，如下所示。

```
$ git checkout master
```

一旦你评审完这些更改，就可以通过变基更新 *master* 分支的本地副本，从而添加新的更改，如下所示。

```
$ git rebase personal/master
```

使用 rebasse 命令可提供一个更干净的图形化历史记录。但是，如果你的团队选择使用合并，那么你可以使用 merge 命令来更新你的本地分支，如下所示。

```
$ git merge personal/master
```

如果你有多个想要更新的分支，则需要单独检查每个分支，并使用相同的流程来合并修改。你需要每次处理一个分支，因为如果两个分支的副本之间出现了冲突，Git 需要给你一个工作目录来解决冲突。

项目的消费者只需要用到上面几个命令。然而，如果开发者在本地略微修改了她的仓库副本，并想要将更改贡献回项目，她只能提交一个补丁，或请求开发者权限（对于一次性贡献者来说不太可能得到授权）。虽然她可以提交补丁，但这不是最好的选择。（是的，有一些项目仍然在使用补丁，包括 Git 项目本身！）相反，很多项目都倾向于使用*拉取请求*。这个术语起源于 GitHub，后来在其他系统上也流行开来。一个拉取请求是一个元功能，它不是 Git 的内置功能，而是 Git 的辅助软件的功能。它为项目维护者提供了一个可视化的提示，以合并远程仓库中的一个工作分支。两个仓库之间的连接只存在于某次特定的请求中。这个连接不是持续的，与开发者在自己的本地工作站中设置的远程仓库连接不同。

7.2.2 贡献者

你认为你有兴趣为一个软件项目做出贡献。太棒了！（作为本书作者，我感到很释然。如果你看到了这里，仍然没有兴趣参与一个软件项目，我会感觉很糟糕。）作为一个分布式版本控制系统，Git 专注于你在本地可以做什么。用于共享仓库之间直接协作的内置工具非常简单，不论你是否拥有项目完整的写入权限。Git 中没有分支单独的权限。事实上，如果不考虑 SSH 支持，Git 中完全没有任何认证系统。Git 依赖外层软件来提供访问控制。

为了使外层软件在两个仓库间建立连接，会要求这两个仓库可以从同一个地方访问到。为此，最简单的方法是让开发者将他们的更改上传到主项目仓库所在的系统。GitHub 和其他基于 Web 的系统都会克隆或派生“项目”，并将你的更改上传到复制的仓库。然后，使用外层软件来请求将你的变更拉取到“项目”仓库。

使用 GitHub 术语的步骤如下所示。

(1) 一个想要做出贡献的开发者（“开发者”）派生了“项目”仓库。
(2) 然后，“开发者”可以在自己的“项目”副本中提出建议的更改。
(3) 完成后，“开发者”发起一个从她项目副本中的一个分支到“项目”仓库的拉取请求。
(4)“开发者”和“维护者”将使用 GitHub 网页上的评论功能来进行讨论。有时，在“维护者”接受提议，更改进入“项目”前，“开发者”需要进行一些更新。
(5) 如果提议的更改对项目有益，“维护者”会将这个拉取请求并入“项目”

GitHub 不需要你将“项目”克隆到本地

现在，GitHub 允许开发者在网页上直接对文件进行简单的编辑。但是，很多开发者选择克隆一份“项目”的副本，以便在本地工作。然后，当他们完成自己的工作后，将更新推送到自己的项目副本，发起一个从自己的项目副本到主项目仓库的拉取请求。

提交拉取请求的流程根据所使用的外层软件（例如 GitHub、Bitbucket、GitLab 等）不同而略有区别。不过，基本的流程会在第三部分中介绍。

7.2.3　维护者

拥有“项目”仓库直接的提交权限的开发者是一种特殊类型的开发者，称为“维护者”。根据你的团队结构，“维护者”可能是质量保证团队内部的成员，也可能是社区中精心挑选的开发者。对于小型内部项目来说，“维护者”可能是正在为项目工作的任何人。

在第 2 章中，你了解了一些关于项目治理模型的知识。“维护者”参与项目的方式是政治决策，而不是技术决策。Git 实际上不关心你的项目结构，所以你需要设计一个最适合你的系统。定义“消费者”和“贡献者”工作流相对容易，因为你并不真正和 Git 打交道，而是使用由外层软件定义的工作流（对于“消费者”来说，他们甚至更不使用 Git）。

如果你的团队中每个人都是“维护者”（即允许他们直接提交到仓库），那么你可以选择是否要求开发者创建一份仓库的单独克隆。唯一的限制是，你的代码托管系统可能不支持合并单个仓库内部的分支。查看你选择的系统，看看它是否推荐使用某种工作流。

我一般和少于 10 个开发者一起工作。我合作过的一些团队选择为每个开发者分配单独的远程仓库，一些团队允许开发者直接将他们正在进行的工作提交到“项目”仓库。在 Drupal 项目中，有成千上万的开发者，只有少数人可以提交到主项目仓库。然而，除此之外还有 30 000 个贡献产生的模块，其中每一个模块的维护者都拥有直接访问项目仓库的权限。

一切以文档中的规则为准

如果没有文档规则，你的项目将会陷入无序的状态。因此，写下你希望人们在为项目做出贡献时所遵循的确切步骤。

项目维护者需要在本地至少有一个“项目”仓库的克隆。如果你是启动项目的开发者，那么已经拥有此仓库的本地克隆。如果不是，你将需要使用以下命令来克隆仓库。

```
$ git clone https://<hosting-url.com>/<project-owner's-name>/<project-name>.git
```

你在第 6 章中学到了如何使用以下命令在单人团队中创建项目仓库的克隆，如下所示。

```
$ git clone https://gitlab.com/gitforteams/gitforteams.git
```

这一步将会创建仓库的本地副本和一个名为 `origin` 的远程连接。

如果你的项目需要，你可能还需要在代码托管系统上创建一份“项目”的克隆。这在上一

节中有所介绍，或者你可以遵循第三部分中提供的更详细的说明。一旦你创建了远程克隆，就可以将这个远程连接添加到你的本地仓库。这将允许你在同一目录中切换本地仓库和远程仓库。如果你喜欢，可以保留两个本地目录，但我个人非常享受不用频繁切换带来的高效率。你可以使用自己的约定来命名远程连接。添加新的远程连接的格式如下所示。

```
$ git remote add nickname https://<hosting-url.com>/<your-name>/<project>.git
```

如果我要从 GitLab 添加我自己的克隆，接着前面的示例，我将使用下面的命令。因为这个连接对应的是我个人的仓库副本，所以我将会使用 *personal* 别名，如下所示。

```
$ git remote add personal https://gitlab.com/emmajane/gitforteams.git
```

为了避免混淆，我可能还会将“项目”的远程仓库从 *origin* 重命名为 *official*，如下所示。

```
$ git remote rename origin official
```

这些别名对你的系统来说是完全随意和私人的。它们不会与其他人共享，因此请使用对你来说有意义的名称。一般来说，我们约定用 *origin* 代表最接近于本地工作的远程副本，而 *upstream* 代表其他开发者添加了最新功能的仓库副本，你可能想要将其并入你自己的工作中。

一旦你设置好了项目和自己仓库副本的远程连接，就应该验证这些名称和 URL 是否符合你的预期，如下所示。

```
$ git remote --verbose
```

我的操作的输出如下所示。

```
official        git@gitlab.com:gitforteams/gitforteams.git (fetch)
official        git@gitlab.com:gitforteams/gitforteams.git (push)
personal        git@gitlab.com:emmajane/gitforteams.git (fetch)
personal        git@gitlab.com:emmajane/gitforteams.git (push)
```

现在，你可以作为“贡献者”和“维护者”在你的项目中工作。

7.3 参与开发

在使用 Git 工作时，你会参与四种主要活动：完成新提出的更改，保持分支最新，审查提出的更改，以及发布完成的工作。在更新分支或试图将提出的更改并入“项目”时，你不可避免地需要解决冲突。

7.3.1 构建完美的提交

有两种基本的提交方法：展示思维过程和呈现最终解决方案。当我在使用一种不太熟悉的编程语言时，就会增量式地思考，每次专注于系统的一小部分。在我工作时，会在关键点前提交我的工作快照。这些快照作为生命线，让我跟踪自己如何思考问题。如果你阅读过我编码时写下的提交说明，就能够轻松地弄明白我的想法。提交可能表示小到 15~30 分钟的工作增量。提交说明不太可能解释我为什么做某事。初始提交可能在代码注释前包含一个文档块，概述我将要做什么，下一个提交可能是为我即将构建的工作而搭建的脚手架，

并且从那里继续。比起每个提交的 diff 中所显示的信息，提交说明并不会带来很大的价值。

当我在进行一个更有信心的任务时，更可能在工作目录中作出激进的更改，而不是微小的生命线提交。然后，当我的工作结束后，我会审视整体的变化，形成更小、相关的提交。这可以通过一次提交一个更改的文件来完成，或者我可以使用 `--patch` 模式来进行更细粒度的提交，每次将一个文件的补丁片段添加到暂存区以准备提交。如果我需要使用 `bisect` 命令来挖掘历史，这些策划的提交以后将对我更加有用。例如，为了使用一个函数，它必须已经在某处创建，因此我可能会选择将函数的创建和使用分隔到两个提交，即使我是一起写的。

我不愿将这两种方法叫作新手方法和进阶方法，但这个说法确实有一定道理。不同的源代码管理系统会用不同的方式来表示项目历史中的提交。Git 在向你显示提交历史记录时粒度非常细，因此，使用微小的提交增量的思维方式会带来混乱且难以处理。这就是为什么我们说，当你熟悉 Git 后，你会更有可能采取第二种方法。

不过，你不必放弃你的微小的提交。你可以使用 `rebase` 来将多个未发布的小提交组合成一份更像第二个版本的历史记录。使用你希望的方式工作，然后重构历史记录，使用有用的方式来储存信息。

重写历史记录

是的，我十分讨厌 Git 允许你重写历史记录，然后告诉你这样做有多么危险。对我来说，这像极了傲慢的历史记录修正主义。但 Git 使用这个模型。为了高效地使用 Git，我抛开自己的成见，采用软件自身认为的最佳实践。我并不害怕变基，只是不喜欢它一开始的存在。我允许你也不喜欢它，但不喜欢不是不去使用它的理由。它深深植根于 Git 如何储存代码历史记录的元数据的原理之中。

如果你不小心在两个提交之间做了太多工作，也可以拥有细粒度的提交历史记录。你已经学过如何将单个文件添加到暂存区。你可以获得更细的粒度，将单个文件中的编辑分配到多个提交。为了添加文件中的部分修改，而不是整个文件，请使用 `git add --patch` *`filename`* 这个命令。这个命令将会逐行遍历你的文件，询问你是否想要在正在构建的提交中包含每个已更改的行。

仅在必要时重写历史记录

如果你有着在中央服务器上显示正在进行的工作的文化，就需要在变基你的工作时特别小心。当一个提交变基后，系统会给任何更改过的提交对象的元数据分配一个新的标识符。例如，如果你在更新一个分支，你的本地提交现在有了新的父提交并获得了一个新的 ID。如果你在试着清理分支的历史，并压缩了两个提交，那么系统会将一个新的 ID 分配给生成的提交对象，即使内容是相同的！这个双重时间线可能会使 Git 感到困惑并导致冲突。为了避免这些冲突，请在可以短暂存在的分支中使用交互式变基，如工单分支。

优秀的提交对象具有如下特征。

- 只包含相关的代码。没有作用域跃迁，没有“只是修复了空白字符的问题”。
- 遵从项目的编码标准，包含行内文档。
- 大小合适。或许是 100 行代码。或许是一个大型重构，更改了一个函数名，以致 1000 行代码受到了影响。
- 用最佳的提交说明描述工作（见下节）。

我听到过关于提交说明最好的经验法则是“不论付出什么代价，保证以后我不会因为过去太懒而感到懊悔”。

你的提交说明应该包含以下内容。

- 使用标准化格式的简略描述（少于 60 字），以便检索日志。
- 一段稍长的解释，说明当前代码为什么有问题，以及为什么你的更改很重要。
- 关于更改如何解决手头问题的高层描述。
- 概述更改可能产生的副作用。
- 所做更改的摘要，以便阅读代码 diff 时能够确认提交说明。但阅读 diff 不是用来猜测所发生更改的办法。
- 工单编号，或指向源码的引用，提议的更改可以、已经或将要在其中进行讨论。
- 谁会受到这个更改的影响（例如针对开发者的优化，或针对用户的改进）。
- 列出文档哪些地方需要更新。

下面显示的是一个不好的提交说明。

```
git commit -am "rewrote entire site in angular.js - it's faster now, I'm sure"
```

因为以下原因，这个提交不够充分。

- 由于使用了 `-a` 参数，所有文件将会作为此提交的一部分提交，却没有考虑是否应该将它们纳入这个提交。
- 由于使用了 `-m` 标志，我们总是倾向于编写简要的提交说明，而不是具体描述为什么这个更改是必要的，以及这个更改如何解决问题。
- 这个提交说明没有引用工单号，因此无法知道哪些问题现在已经解决了，并且可以在工单跟踪工具中关闭。

作为对比，一份好的提交说明应该如下所示。

```
$ git commit

[#321] Stop clipping trainer meta-data on video nodes at small screen size.

 - Removes an unnecessary overflow: hidden that was causing some clipping.

Resolves #321
```

由于以下原因，这是一份好的提交说明。

- 它在精简的提交说明开头用方括号包含了工单号，以便以后更容易地读取日志。
- 精简的描述（用于精简日志视图）解释了站点访问者看到的现象。

- 消息的说明解释了用于解决这个问题的技术实现。
- 提交消息的最后一行（`Resolves #321`）将会被工单系统捕捉到，并将工单的状态从“进行中”（open）移动到“需要评审”（needs review）。

在提出更改时，你应该保持提案精简，专注于解决单个问题。这将使项目的“维护者”更容易地审查你的提交，并且接受你的工作。例如，如果你在修复代码库某处的一个特定 bug，不要修复代码其他地方多出来的一个行尾。虽然项目可能有自己的分支命名约定，如果你在贡献一个偶然发现的修复，而这个修复还不是一个“项目”仓库中已识别的问题，请使用你正在解决的问题的精简描述来命名你的分支，例如，*css_button_padding*（按钮的 CSS 内间距）或 *improved_test_coverage*（提高测试覆盖率）（例 7-3）。

例 7-3　向代码库提交更改

```
$ git checkout -b terse_description
(edit files)

$ git add filename(s)
$ git commit
```

此时，提交说明编辑器将会打开，你将需要提供你曾写过的最好的提交说明。

当提议的更改准备完毕后，你现在可以使用 `push` 命令将它推送到你的仓库副本，如下所示。

```
$ git push
```

你的个人分支已经上传，现在是和队友一起将你的更改并入项目主分支的时候了。

7.3.2　保持分支最新

储存在 Git 中的分支通常被视为以下两者之一：官方的项目分支和短期存在的建议分支。共享的项目分支被用于集成由多个开发者评审并通过的代码，且包含项目代码的官方历史。你的这些分支的本地副本应该永远保持最新并应该总是被用作你的工单分支的基线分支。按照惯例，将新的更改写入一个官方分支的本地副本是不合适的。相反，你应该创建一个新的分支并完成你的工作，然后将那个分支合并回官方分支。在第 3 章中我们讨论了几种分支策略，如果你的团队还没有选择一个分支策略，你可能会希望回过头再看看那一章。第二种类型的分支事实上是开发者的沙盒。在其中你可以测试新的想法，并让你的代码为评审作好准备。这些短期存在的工作分支也必须保持最新，但需要使用一个略微不同的方法。

问题又来了，是选择变基还是合并？

现有仍然不会有任何变基警察突然出现在你的团队会议上。你需要明白，作为一个团队，你们将如何解决分支更新的问题。（我仍然认为你需要做的是对你的团队最好的事，但我将向你展示变基的指示，这样你会发现使用这种方法并不是很难。）不管你使用哪种方法，仔细记录你的解决方案，并且帮助 Git 新手，确保他们能够一模一样地执行这些命令。我发现确保一致性最好的办法是提供方便复制和粘贴的文档，并且让人们在命令行中工作。此外，流程图也会很有用（http://gitforteams.com/resources/merge-rebase.html）。

为了减少合并短期存在的分支时需要处理的冲突数量，你应该保持你的工作分支与你即将并入的项目分支同步。“定期”是多久？我建议你更新你的分支的频率，至少和你喝咖啡的频率一样。如果你不喝咖啡，我会建议你至少每天使用例 7-4 中的命令更新一次你的工作分支。是的，这看起来很乏味，但从长远来看，保持你的工作尽可能最新可以节省你很多时间。

例 7-4　更新项目分支的本地副本

```
$ git checkout master
$ git pull --rebase=preserve
```

Git 会更新主仓库的本地副本来并入上游仓库的更改。

一旦项目分支是最新的，你现在可以更新你的工作分支了。然而，如果你要更新的工作分支与共享的项目分支不同，那么你就不会有一个上游分支可以让你拉取更改。因此，你如何知道你现在是否应该合并或变基？经验法则如下：你刚开始的工作是否会被立刻并入已存在的工作分支？如果它是你写的一个功能，它不会出现在你刚刚更新的分支上，因此你应该合并这个分支来并入新的工作。如果它是其他人写的一个功能，你几乎一定会想要变基（如果你的团队使用变基）。另一个有用的小技巧是匹配名称。如果你想要并入的更改来自一个同名但在不同远程仓库上的分支，那么你几乎一定想要变基。

在 Git 中，变基和快进合并都会导致线性的时间轴，因为它们重演了你在其他分支中提交的工作。由于每个提交都被重演了，存在需要你解决合并冲突的可能性。因此，对处理合并冲突不那么自信的开发者可以选择简化流程，使用 `merge` 命令来更新他们的工作。使用 `merge` 确实使你的历史记录更难阅读，但在技术上更简单了，因为它通常涉及更少的合并冲突。

如果你在一个复杂的代码库上工作，能够快速运行调试工具非常重要，你应该多花点时间使用 `rebase` 命令来更新你的工作分支，从而获得一个干净的历史记录。但是，如果觉得使贡献尽可能简单更为重要，你可能会希望允许你的开发者使用 `merge` 命令来更新分支。（最 Git 的 Git 读者在看到这里时会恨得咬牙切齿。但你知道吗？即使你的团队决定想要更简单的方法，也不会有 Git 警察突然出现在你的门前。我保证。在这里放张蜜獾的图片不太合适，让我们继续讲下去吧。）

在更新你的工作分支时，你要做的第一件事是确保你的项目分支是最新的。保持一个共享分支最新的方式通常是使用 `pull` 命令（使用可选参数 `--rebase`）。为了更新你的个人工作分支，你需要记住你最初起源的源分支并将这个分支上的更改复制到你的工作分支。如果你遵循第 3 章中描述的 GitFlow 模型，这个分支可能是 *dev* 或 *development*。

例如，如果你的工作分支名为 *2378-add-test*，而你的源分支名为 *development*，则命令如下所示。

```
$ git checkout development
$ git pull --rebase=preserve
$ git checkout 2378-add-test
$ git rebase development
```

你在工作分支中做出的每个提交现在将被重新应用，看上去来自 *development* 分支的新提

交一直就在那里。这些提交可能会干净地应用，或者你可能需要处理合并冲突。因为变基是 Git 中更新分支首选的方法，所以我将不得不向你省略如何合并分支的命令。希望你会原谅我。

除了保持你的分支最新之外，在你自己的工作被并入“项目”后，你还应该记得更新你的个人仓库，因为“项目”的主分支现在包含了新的提交。当你负责审查别人的工作并将其合并到 `master` 分支时，这将非常有用。你允许的命令与之前描述的完全相同，如下所示。

```
$ git checkout master
$ git pull --rebase=preserve
```

不管你选择如何更新分支，我希望你至少尝试在工作流中使用变基。尽管有时它会令你沮丧，但如果你需要使用第 9 章中描述的调试技术，那么变基能够帮助你获得一个更干净的历史记录。

7.3.3　评审工作

为了评审别人的工作，你必须在你自己的仓库中获得一份该工作的本地副本。这可能是已经并入官方项目分支中的工作，也可能是同事请你评审并合并到主项目的一个新功能或 bug 修复。

同行评审新的工作是一个多步骤的过程，第 8 章将对其进行更详细的讨论。基本过程如下所示。

(1) 添加一个到相关仓库的远程连接。
(2) 从那个仓库中获取可用的分支。
(3) 为你想要仔细检查的分支创建一份本地副本。
(4) 将其他分支中你想要的更改并入你的工作。
(5) 将修订后的分支推送回相关的远程仓库。

你要做的第一件事是找到包含你想要并入的工作的仓库。使用 `remote` 的子命令 `show` 列出每个远程仓库（例 7-5）。和列出分支一样，所有可用的远程都会列为命令的输出。在例 7-5 中，我在上一节中添加的两个远程显示了出来。这会快速提醒我想要深入研究哪个仓库。

例 7-5　远程仓库的精简列表

```
$ git remote show

official
personal
```

一旦你有了仓库的名称，就可以在上个命令中添加别名来获得完整的远程列表（例 7-6）。

例 7-6　远程仓库 `personal` 的全部细节

```
$ git remote show personal

* remote personal
  Fetch URL: git@gitlab.com:emmajane/gitforteams.git
  Push  URL: git@gitlab.com:emmajane/gitforteams.git
  HEAD branch: master
  Remote branches:
```

```
  2-bad_jokes   tracked
  master        tracked
  sandbox       tracked
  video-lessons tracked
Local branch configured for 'git pull':
  master merges with remote master
Local ref configured for 'git push':
  master pushes to master (up to date)
```

这里我可以看到远程仓库中储存了四个分支，每个分支我在本地都有一个对应的副本［由单词 tracked（跟踪）指示］。

更新你的本地分支列表

如果你已经有了一个远程仓库的连接，而且没有看到你的同伴让你评审的分支，请首先运行 `git fetch` 命令，确保远程分支列表是最新的。

如果你不希望获得所有关于远程仓库的信息，那么可以选择使用 `branch` 命令和 `--remotes` 参数来只显示远程分支（例 7-7）。这将使你能够定位到你需要评审的分支。与参数 `--all` 相比，我更喜欢这个 `branch` 的变种，因为它给出了分支的真名，而不会添加 `remotes` 引用信息。

例 7-7　列出远程分支

```
$ git branch --remotes
```

分支是提交的组合

分支是一个线性的开发过程，将单个提交对象连接起来。不同的分支实例上可能有不同开发者创建的提交，在仓库同步前并不相同。这其实是一种混沌的状态，但混沌只存在于每个小仓库中。作为软件团队，我们建立的约定是为混沌带来秩序，并允许我们以一种理智的方式共享我们的工作。还记得我们在第 3 章中学到的分支策略吗？它们将工作按照逻辑思维流组织。还记得第 2 章中的权限策略吗？它们会把人们锁定在正确的仓库中，如果没有社区看守者的帮助，无法进行更改。

如果你为 `branch` 命令添加了 `--verbose` 参数，分支顶端的单行提交说明将会被包含在输出中。例如，我有多个活跃的工作分支、一个集成分支和一个项目的官方分支（例 7-8）。尽管我偶尔把我的提交上传到远程服务器，大多数时候我只是在章节分支上工作，将我的工作并入集成分支 *drafts*，然后并入主分支 *master*。

例 7-8　在编写本章时，`git branch --verbose` 的节选输出

```
  ch02    7313755 CH02: Adding patching workflow diagram.
  ch04    69a3ded CH4: Stub file added with notes copied from Drupalize.Me.
* ch05    80b5200 [official/ch05: ahead 2] CH05: Fixing URL for image 05fig01.
  drafts  80b5200 CH05: Fixing URL for image 05fig01.
  master  319bb53 [official/master] Merge branch 'drafts'. Updates for CH05.
```

第一列包含分支名，第二列包含提交 ID，第三列包含最新的提交消息的首行。如果一个分支是远程跟踪分支，远程分支的名称出现在提交 ID 和提交说明之间的方括号中。

一旦你找到了包含你想要评审的工作的远程分支，就可以将分支复制到你的本地仓库（例 7-9），或者使用 `log` 和 `diff` 命令来检查对它的应用（例 7-10）。

例 7-9　将一个远程分支复制到你的本地仓库

```
$ git checkout --tracking remote_nickname/branch
```

例 7-10　在不创建工作副本的情况下检查一个远程分支

```
$ git log --oneline remote_nickname/branch
$ git diff current_branch...remote_nickname/branch
```

假设工作通过了审查，那么就该将它合并到主项目分支了。

7.3.4　合并完成的工作

在将新工作合并到你的项目分支前，你需要确保所有分支都是最新的。这一步是必要的，因为如果（远程上的）目标分支包含不在你本地副本中的提交，Git 将不允许你推送远程仓库的副本。

在将新工作上传至远程服务器时，Git 只会接受快进合并的工作。这意味着你不必担心在推送你的工作时遇到合并冲突。由于这个限制，你的本地分支需要包含所有远程提交，然后才能推送你的分支。为了更新你的工作，你需要使用 `pull` 命令来获取所有远程服务器上的更改，并将任何新工作并入你的本地分支。

首先，使用带 `--rebase` 参数的 `pull` 命令来更新你的目标分支副本（例 7-11）。

例 7-11　并入项目分支中的更新

```
$ git checkout master
$ git pull --rebase=preserve
```

在公共分支更新后，你还需要更新功能分支（例 7-12）。

例 7-12　将一个完成的工单分支合并到一个公共的项目分支

```
$ git checkout 2378-add-test
$ git rebase master
```

最后，你可以将工单分支并入主项目分支（例 7-13）。

例 7-13　将完成的工单分支合并到公共的项目分支

```
$ git checkout master
$ git merge --no-ff 2378-add-test
```

如果所引入的更改和以前完成的工作不同，则合并即将完成；但是，如果同一区域中有重合的工作，Git 将会不知道如何完成这个合并，会寻求你的指导。在 Git 术语中，寻求帮助通常称为合并冲突，听起来有些吓人。

7.3.5　解决合并和变基冲突

冲突听起来就很棘手和可怕，但在 Git 中，合并冲突事实上是一个非常小的问题，你不需要花很多钱请一个协调人或者治疗师来解决它。在任何时候，文件都是在原地被修改的，Git 不确定哪个版本才是正确的版本，因此会请你来做这个决定。Git 将这种不确定性称为冲突。

当你合并两个分支时，总有可能出现这样的情况，我们和他们同时修改了同一个文件中的同几行代码。

Git 会在任何文件的冲突位置添加下面三行冲突信息。

```
<<<<<<<
=======
>>>>>>>
```

我们的代码和他们的代码通过一行 = 分隔开来。为了解决一个冲突，你将需要编辑这些文件，选择要保留的正确内容并删除标记。当你打开文件来查看冲突时，顺便检查一下旁边的区域。有时 Git 会弄错应该放置标记的地方，因此你不应该简单地删除其中一整块区域，或另一块区域。仔细阅读，当你看到下面的代码时，会发现你需要保留每一块区域中的一部分内容。

```
<<<<<<< HEAD
    $p++;
}
=======
}

>>>>>>> 2378-add-test
```

如果不理解这里代码更新的目的，我们就没有足够信息来解决这个合并冲突。或许右括号应该保留，因为它同时出现在了冲突的两侧，但新增的那一行呢？自增的变量呢？如果你遇到了不知道如何解决的合并冲突，无法直接通过阅读代码来弄明白，你应该与代码的原作者讨论。如果你错误地解决了一个冲突，误解代码并删除了过多（或过少）代码可能会导致意外地引入新的 bug。

一步步解决两个分叉的分支间的冲突

有一个辅助程序，git-imerge（https://github.com/mhagger/git-imerge），用于合并两个分支顶端的提交。使用增量式提交使你更容易看清冲突应该被如何解决，因为每次只需要更少的内容。这不是 Git 核心的一部分，你需要单独下载并安装这个软件。如果你想要减少安装的麻烦，请查看你最喜欢的包管理工具。我的副本是通过 OS X 的 Brew（https://brew.sh/）安装的。

当你完成编辑后，可以移除 Git 放到文件中的标记，使用屏幕上 Git 状态说明提供的指示继续操作，如下所示。

```
$ git status
```

如果你完成了一个合并，就需要添加更新后的文件，并将它们提交至你的仓库，如下所示。

```
$ git add filename(s)
```

通过每次添加一个文件，你可以将 status 命令用作一个待办列表，列出包含合并冲突并需要进行处理的文件，如下所示。

```
$ git status
```

一旦每个文件中的合并冲突都被清理完之后，你就可以提交你暂存的更改，如下所示。

```
$ git commit
```

此时，Git 默认的文本编辑器会打开，同时显示关于你正在完成的提交的额外信息。当你完成了说明的编写后，保存更改并退出编辑器以继续。

如果你在遇到合并冲突时尝试变基，或许会进入一个多步过程中。在这种情况下，你需要继续变基过程，如下所示。

```
$ git rebase --continue
```

如果在开始合并前，你深信不疑你想要全部使用引入的工作（他们的）或者你自己的工作（我们的），那么可以预先告诉 Git 你希望如何处理两个分支中的提议更改。例如，如果你想要合并一个分支，并知道其中包含了一个问题的修复，就可以强制 Git 使用另一个分支来更新你自己的分支，如下所示。

```
$ git checkout branch_to_update
$ git merge --strategy-option=theirs incoming_branch
```

7.3.6　发布工作

在你第一次向指定分支上传你的更改时，需要指明想要使用的远程仓库和分支名。习惯上我们会保持本地和远程仓库中的分支名一致。你将需要包括远程仓库的别名。在例 7-14 中，我们假设远程仓库的名称为 *origin*。

例 7-14　将包含提议更改的分支上传至远程仓库

```
$ git push --set-upstream origin branch
```

一旦你在远程仓库中建立了这个分支，就可以使用 push 命令将工作再次上传到同一个远程仓库，如下所示。

```
$ git push
```

如果你的仓库拥有多个远程仓库，那么你将需要显式地将更改分别推送到每一个远程仓库。默认情况下，我们使用 *origin*，如下所示。

```
$ git push remote_nickname
```

这个过程的下一步取决于你使用的托管系统。不过，一般来说，前往“项目”页面，你可以在上面找到一个*拉取请求*（pull requests）链接（这个表述在你选择的系统中可能会略有不同）。通过这个链接，你应该能够发起一个请求，将你提议的更改纳入项目。系统应该已经知道你的哪个仓库是克隆自“项目”的，并且应该列出你在副本中工作过、可能包含“项目”提议更改的分支。你将选择想要并入的分支，并完成任何其他所需的工作。第三

部分深入讨论了这个流程。

一旦你的提交请求提交后，“维护者”将审查你提议的更新。他可能会按照原样接受你的工作，或要求你修改并重新提交你的工作。如果需要其他更改，重复本节中列出的步骤，直到该拉取请求被接受。

为了将新工作发布到共享分支上，你要做的第一件事是检查你将要合并的分支是否是最新的，这将确保你可以在合并更改后推送你的工作。如果这个分支不是最新的，你将无法上传这个共享分支的修订副本，直到你下载了新更新并将其并入该分支为止，如下所示。

```
$ git checkout master
$ git pull --rebase=preserve
```

在你的主项目分支的本地副本更新之后，你应该确保这些更改被同时复制到你所在的功能分支中，使得在执行合并之前两个分支的差异最小，如下所示。

```
$ git checkout 2378-add-test

$ git rebase master
```

在工作分支更新后，你可以合并已通过评审且被接受的更改，如下所示。

```
$ git merge --no-ff 2378-add-test

$ git push
```

现在，你可以从你的本地仓库和任何你拥有写入权限的远程仓库中删除这个工作分支，如下所示。

```
$ git branch --delete 2378-add-test
$ git push remote_nickname --delete 2378-add-test
```

你的分支现在应该已是最新，可供你的同事下载。

接下来发生的事情很大程度上取决于你正在构建的软件类型。想要连接 Git 和持续集成构建服务器的 Web 开发者将会受益于 Lorna Mitchell 的视频 Git Fundamentals for Web Developers（http://shop.oreilly.com/product/0636920042129.do）。

7.4 样例工作流

本章剩余部分是与团队一起工作时的模板。你应该与你的团队讨论他们想要如何工作，并写下每个贡献者和维护者在项目期间需要使用的命令。

7.4.1 基于冲刺的工作流

这个流程与我在多个基于冲刺发布周期的团队中使用的流程基本相同。这是一个 GitFlow 的变体，适用于网站的每周部署。冲刺的安排遵从单周（相对于更“传统”的双周冲刺）。这会鼓励细粒度的工单并帮助开发者尽快在产品中看到他们的工作。如果一些工单范围较广，它们可能需要在几个“冲刺”中完成。

仓库设置有五种不同类型的分支：开发分支、工单分支、QA 分支、主分支和补丁分支（表 7-1）。这些分支要么用作单个问题的开发分支，要么用作集成分支。

表7-1：每周部署工作流中的分支类型

分支名/约定	分支类型	描述	基线分支
dev	集成	用于放置通过同行评审的代码	工单分支
ticket#-descriptive-name	开发	用于完成工单中标识的工作	*dev*
qa	集成	用于每个冲刺结束前的质量保证测试，没有通过 QA 测试的代码会从这个分支上移除	*dev*
master	集成	用于部署测试全部通过的代码	*qa*
hotfix-*ticket#-description*	开发	用于为产品中发现的紧急问题开发对策	*master* 上最新的发布标签

对于开发者来说，每天都是开发日。此外，每周有三天所有团队成员为同一个目标奋斗。

对开发者来说，这个工作流并没有过于复杂（例 7-15）：所有的工作都始于一个全新的、继承自父分支 dev 的工单分支。一旦完成后，工单分支中的工作被推送到共享项目仓库。分支通过变基保持最新，与合并相比，你会得到一个更干净的分支历史记录。

例 7-15　完成工单时需要的 Git 命令

在这个例子中，用远程的名称替换 *origin*，用工单分支的名称替换 *1234-new_ticket_branch*，如下所示。

```
$ git checkout dev
$ git pull --rebase=preserve origin dev
$ git checkout -b 1234-new_ticket_branch
// 完成工作
$ git add --all
$ git commit
```

在共享工作前，确保该分支包含了新的提交，如下所示。

```
$ git checkout dev
$ git pull --rebase=preserve
$ git checkout 1234-new_ticket_branch
$ git rebase dev
```

最后，与其他人共享新的工作，如下所示。

```
$ git push origin 1234-new_ticket_branch
```

在完成后，工单分支会由团队中另一个人评审（例 7-16）。如果代码通过了评审，评审者将工单分支合并到开发分支并从主仓库中删除这个工单分支。第 8 章中深入讨论了这个评审过程。

例 7-16　完成同行评审时需要的 Git 命令

```
$ git checkout dev
$ git pull --rebase=preserve
```

```
$ git checkout 1234-new_ticket_branch
// 进行评审工作
$ git merge --no-ff 1234-new_ticket_branch master
$ git branch --delete 1234-new_ticket_branch
$ git push --delete origin 1234-new_ticket_branch
```

质量保证（周一至周二）的工作内容如下所示。

- *dev* 分支上运行着自动化的测试集，用于捕获新加入的功能分支中可能引入的任何回归。
- *dev* 分支上的所有工作都被合并到 *qa* 分支进行测试（例 7-17）。开发工作在 *dev* 分支上继续。
- 在共享文档（如 Google Docs）中创建一份冲刺列表，将并入 *qa* 分支的工单中的用户故事复制到文档中。一般来说，这是工单描述的第一行，你应该采用这个约定来加速 QA 的流程。
- 所有团队成员有义务完成共享文档中工单列表的测试。除了每周的工单之外，可能还需要有人来完成滚动更新的测试。
- 为任何没能通过质量保证测试的问题创建一个新的工单，使它可以在发布前进行修复、撤销（例 7-18）。

例 7-17　设置 *qa* 分支时需要的命令

```
$ git checkout dev
$ git pull --rebase=preserve
$ git checkout qa
$ git merge --no-ff dev
$ git push
```

例 7-18　移除在发布前没能通过 QA 的工单时需要的命令

```
$ git log --oneline --grep ticket-number
(找到需要撤销的提交)

$ git revert commit

$ git revert --mainline 1 merge_commit
(但是,在理想情况下,你应该使用--no-ff合并工作分支,这样会产生一个便于撤销的提交ID)
```

发布日（周三）的工作内容如下所示。

- *qa* 分支被合并到 *master* 分支并添加标签（例 7-19）。
- 在线上更新仓库，使用为发布准备的带标签的提交。
- 为开发团队下周的工作安排优先级。

例 7-19　准备部署时需要的命令

```
$ git checkout master
$ git merge qa
$ git tag
(找到最新的标签,以确定下一个标签的编号)

$ git tag --annotate -m tag_name
$ git push --tags
```

在标签添加之后，它通过 `--annotate` 参数签署，通过 `-m` 参数添加说明。这确保了这个标

签不会被忽略。

公布日（周四）的工作内容如下所示。

- 向社区用户公布前一天上线的更改。多出的一天使得团队有机会处理代码移动到生产环境后出现的意外回归或 bug。
- 在前一天新建的优先事项列表上继续开发。

如果出现了不太可能发生的情况，一个严重的 bug 或回归被引入了生产环境后，一个补丁随之产生。当然了，严重是一个相对的词。在这个系统中，每周进行一次部署，而补丁一般无法等到一周才进行部署。

每次部署都被打上了标签，因此你的第一步是获取全部标签列表，并定位到代码库中当前活跃的版本（例 7-20）。从此处创建一个新的分支，应用更新的代码，并在部署前上传，等待评审。

例 7-20　创建补丁分支时需要的命令

```
$ git checkout master
$ git tag
（查看标签列表，确定当前活跃的标签）

$ git checkout -b hotfix-issue-description tag_name
```

接下来，补丁分支将被当作常规的开发分支一样对待，经过同行评审和质量保证测试。当测试通过后，它会被立即合并回 `master` 分支并打上标签准备发布（例 7-21）。

例 7-21　准备补丁的部署时需要的命令

```
$ git checkout master
$ git merge --no-ff hotfix-issue-description
$ git tag --annotate -m new_tag_name
$ git push --tags
```

在这个系统中，没有使用语义化的版本命名。相反，标签名称是使用格式 *<launch_version>.<sprint_week>.<hotfix>* 进行递增的。例如，1.4.3 将会被用于表示开发中第四周的第三个补丁（换句话说：这个团队这周糟透了！）。

7.4.2　没有同行评审的可信开发者

在写这本书时，我使用的是 O'Reilly 的自动化构建工具 Atlas（https://atlas.oreilly.com/）。这个系统还提供了一个网页，使得编辑可以直接在上面操作图书文件，保存的文件被立即提交到 `master` 分支。界面上没有同行评审流程，因为我的团队中任何人都可以直接编辑一份文件。然而，我偏好在本地工作，而不是通过网页。最初，我一直在 `master` 上工作，本地的分支开销很低。直到一次出现的合并冲突改变了我在本地的工作方式。

当我想要更新我的工作时，我会使用 `fetch` 命令来查看我的编辑做出的修改。通过 `fetch` 命令，我将我的 `master` 分支的副本与他们的 `master` 分支的副本（`origin/master`）相比较。假设我同意编辑做出的所有修改，我会把他们的分支副本并入我本地的副本；如果我有异议，我会使用策略 `ours`，即丢掉他们的修改，但是让 Git 以为这两个分支已是最新，如下

所示。

```
$ git checkout master
$ git fetch origin
$ git diff origin/master
```

根据我是否想要保留这些更改，我会使用三种方法之一来合并工作：合并所有工作、用他们的工作覆盖我的或用我的工作覆盖他们的。

若想合并所有工作（真正的合并），请执行以下命令。

```
$ git merge origin/master
```

若想保留我自己的工作，请执行以下命令。

```
$ git merge -X ours origin/master
```

若想舍弃我自己的工作，以满足评审者的要求，请执行以下命令。

```
$ git merge -X theirs origin/master
```

你可以在提交层面完成合并，抑或在出现合并冲突时，你还可以使用合并工具对层面的合并进行细粒度的逐个更改。（将东西直接丢掉听上去有些像被动的攻击，但这只是单分支系统的限制，你无法在单独的分支中讨论提议的更改。）根据提交的粒度，我或许还会选择拣选一些想要保留的提交，并丢弃其他提交。第 6 章讨论了拣选提交。

最后，我会将本书的新版本上传至仓库，并更新我本地的工作分支 `drafts`，代码如下所示。

```
$ git push origin master
$ git checkout drafts
$ git rebase master
```

然后你开始收到 PDF 格式的评审意见，我又一次意识到，我还有另一种分离工作的方式。我想要新写一章，并保持这些提交干净整洁，但有时候我一章正写到一半，进来了一份编辑修改，我希望及时处理。与其混杂这些提交，我设置了如下的分支结构：`master`、`drafts`，以及每章单独的一个分支，如下所示。

```
$ git checkout ch04
// 编写这一章
$ git add ch04.asciidoc
$ git commit
$ git checkout drafts
$ git merge ch04
```

`drafts` 分支让我整合我正在进行的所有工作。它通过合并已完成的章节来保持同步，或是在编辑进行修改后变基 `master` 分支。当我自己编写章节，没有其他人的贡献时，多分支会产生高昂的维护成本，但当越来越多的贡献者开始提供不同类型的贡献时，更细的分支粒度使我能够选择如何更新我的稿子。

7.4.3　需要独立质量保证的不可信开发者

如果你的团队中大多数都是可信的开发者，但也有一些合约雇员，你可能会希望你的合约

雇员在派生仓库中工作，而不是授予他们主项目的写入权限。在一些类型的软件中，甚至你的员工之间也有这样的分割。例如，如果你在为一个医疗设备开发固件，你可能需要遵守非常严格的政府规定，限制工作雇员的权限提交，在工作添加至仓库前确保评审的执行。

这个模型与本章前面描述的“贡献者”（而不是“维护者”）的模型相同。

第二个例子是在第 2 章的派生策略的描述中给出的。这里我包含了一个如何为 reveal.js 项目提交补丁的描述。为此，我派生了一个项目，然后克隆项目，以便我可以在我的工作站编辑文件。然后，我调转方向，通过推送上传我的工作，然后发起一个拉取请求来提交评审，以将我的更改推送回原始项目。

根据你目前读到的内容，将这些工作流所需的命令放到一起。提示：在本章中，你已经读完了所有内容。首先，自己画一张图，然后添加箭头来显示整个过程中工作的推进。最后，在每个箭头上加上 Git 命令。

7.5 小结

在开始一个新的项目前，你必须首先决定项目的治理结构。这将告诉开发者，他们是需要创建一份项目的远程克隆，还是只需要一份本地克隆。“消费者”“贡献者”和“维护者”的设置权限可能会组织他们进行一些任务，然而，通过添加远程仓库连接，你可以轻松地将一个“开发者”提升为“维护者”。

第 8 章

准备评审

在我长大的过程中，我从父母那里学到了两种评价的方式。父母的回应总是可以预料的。其中一位给你一大堆赞扬的评价，而有着皇家艺术学院学位的另一位则会用设计评判的标准来要求我。我实话告诉你，这一天我既渴望又害怕评审过程。

不幸的是，开发者在学校中几乎没有接触过同行评审过程。一般的评审过程是将作业的最终稿交给导师，而并没有留出讨论如何改进的余地。这种方法没有教会学生根据反馈不断提升。刚开始工作的毕业生可能会默默地嘲笑身边粗制滥造的工作却不加以评价，以至于已经来不及做出更改。

完成一次同行评审是非常耗时的。在我引入强制同行评审的最后一个项目中，我们预计每个工单需要两倍的时间来完成，对于开发者来说，它带来了更多的上下文切换，在等待代码评审时保持分支同步也是带来困惑的根源。然而，它的回报是丰厚的。初级程序员可以接触到代码库中更多的代码，而不只是他们正在进行的那部分工作。资深开发者有更多的机会来质询代码库中可能影响到未来工作的决策，通过采用及时的同行评审过程，我们节省了每个冲刺最后的人工质量保证测试所需的时间。我们认为这些好处值得你投入时间。

8.1 评审类型

在项目生命周期中，应该进行几种类型的评审。尽管本章大部分内容集中于同行代码评审，但你也应该了解其他类型的评审，确保你没有过早（或过晚）评价项目的各个方面。以下是一些评审类型。

- *设计评判*

  通常开发者不参与项目的这个阶段。然而，听取开发者的意见或许会导致用户界面微调，从而极大地降低构建的难度。

- *技术架构评审*
  关于即将构建的底层代码的同行评审。在这个阶段中，开发者应该确保数据模型已经完成，且易于对接构建的所有部分，可能还有未来的功能。
- *自动化自检*
  类似于拼写检查，但是自动化自检是为代码而设计。自动化的自检使得开发者确保他们的代码遵循了项目的编码标准。你可能有其他测试集要运行。这种类型的评审目的是为了所有机器能够轻易自动化检查并捕捉住问题的评审，而不是浪费时间执行人工检查。
- *基于工单的同行代码评审*
  本章大部分内容将会用于讨论这种类型的评审。
- *质量保证/用户验收测试*
  在代码评审之后，新功能将被合并到开发分支，并交由人类测试员测试。这个用户界面评审通常在专门的非生产服务器上进行。

8.2 评审者类型

根据你的项目规模，你可能使用下列几种评审过程之一的变体（或者多种过程的组合）。

- *同行评审*
  我们完全平等地审查代码并将它纳入项目。我们互相学习，在知道同行会随后评价自己的工作之后尽我们所能来完成它。
- *自动化的把门人*
  我们的代码有着测试覆盖率。我们相信我们的测试，并且只提交我们认为会通过大多数测试集的工作。一般来说，我们在代码进入测试集（准备自动化部署）之前，会听取别人的意见。
- *共识的牧羊人*
  我们的程序员社区是警觉、有主见的。在将代码标记为通过社区评审之前，我们需要利益干系人之间形成共识。我们或许还可以在社区中加入一个测试机器人，让人类程序员知道什么时候需要修改以满足最低的要求。
- *仁慈的独裁者*
  我的代码我做主。欢迎你提交你的建议，但我或者我的同事会仔细评审你的工作。我享受寻找你的错误的过程，并可以拒绝你的工作。只有完美才足够好。

同行评审不应该只受限于团队中相同等级的成员。它适用于不同技能水平的组合（表8-1），获益将会有所不同。

表8-1：初级评审者和资深评审者获得的收益

	初级评审者	资深评审者
初级评审者	寻找 bug；遵循标准	学会阅读优秀的代码；建议如何精简；接触到完整的代码库
资深评审者	建议新的技术；改善架构	改善架构；跨功能团队（接触更多代码）

8.3 用于代码评审的软件

本章中概述的命令可以用于任何 Git 托管系统。代码托管系统的详细说明在第三部分中概述，包括使用 GitHub（第 10 章）、Bitbucket（第 11 章）和 GitLab（第 12 章）的说明。这些系统通过*拉取请求*（pull request）或*合并请求*（merge request）来管理代码审查功能，而这些功能相对轻量，使其易于使用并集成到大多数工作流中。

如果由于行业规定，你的汇报要求更加严格，那么你可能需要考虑使用更正式的代码审查和签发流程。下面的软件包专注于代码评审和签发。它们适用于超大型软件项目的代码评审，可能超出了一般项目的需要。

- Gerrit（https://www.gerritcodereview.com/）
  用于 Android、OpenStack 和 Typo3，这个评审系统尤其适合非常大的项目。Dave Borowitz 有一个很不错的关于它的设计（和缺点）的视频演讲（http://git-merge.com/videos/git-at-google-dave-borowitz.html）。
- Review Board（https://www.reviewboard.org/）
  用于 LinkedIn、Apache 软件基金会和 Yelp，这个软件提供了代码何时在代码库中移动的额外信息。

除了手动的代码同行评审之外，它还能帮助开发者进行自动化测试，在发起同行评审请求之前检查他们的工作。一些开源项目，如 Drupal，使用工具来验证代码遵守了编码规范 Coder（https://www.drupal.org/project/coder）。还有一些付费服务，如面向 Ruby 的 PullReview（https://www.pullreview.com/）和面向 JavaScript 的 bitHound（https://www.bithound.io/），它们特定于语言，但与项目无关。

虽然我们专注于技术代码评审，但越来越多的非技术评审员正通过定制的、按需的构建服务器加入评审的环节。一个公开的例子是 Drupal 的 SimplyTest.Me（https://simplytest.me/）服务。这个平台允许人们一次用 30 分钟来部署一台测试机器，其中包含特定的代码补丁，这样他们就可以评审 Drupal issue 队列中的提议更改。这些构建服务器也可以使开发人员受益。审查者不必一个个地进行评审，而可以同时启动多个评审所需的构建流程。现在评审者可以通过同时运行所有需要完成的评审的构建过程，以避免为每个需要完成的代码评审构建本地环境所需的（有时会很冗长）过程。如果这点听上去很吸引你，你应该阅读 Lullabot 的文章“使用拉取请求构建工具”（https://www.lullabot.com/articles/github-pull-request-builder-for-drupal）。如果你的技术栈与他们有些不同，在网页中搜索“拉取请求构建工具”应该能为你指出正确的方向。

8.4 评审issue

在开始本地的代码评审流程之前，你应该在团队的 issue 跟踪工具中仔细阅读提议更改的说明，以了解提议这个更改的原因。这是一个 bug 修复吗？软件遇到了什么问题？它增加了一个新的功能吗？这个功能能够（以何种方式）帮助到谁？在你看代码之前理解这些问题将会帮助你在必要的时候回答“这个代码是解决这个问题的最好方法吗”这个问题。

探索你的代码托管平台

大多数代码托管系统还提供一个 Web 界面，允许你轻松地在线评审提议的更改。在设置你的本地环境之前，使用这个界面快速评审代码。例如，如果提议的更改只是添加缺少的一行代码注释，或者修复一个拼写错误，你可以在线评审这些提议的更改，而无需将所有内容下载到你的本地环境。

一旦你对代码的作用有了很好的理解，那么是时候设置你的本地开发环境了，以便你可以复现“过去”的状态。换句话说，如果这是一个 bug，你应该确保你可以在你的测试环境中复现这个 bug。如果这是一个新功能，你应该确定这个功能尚不存在（不过，两个人不太可能会去实现完全相同的新功能）。

评审别人工作的第一步是验证代码目前是怎么工作的。如果你在测试一个特定 bug 的修复，这意味着你应该从复现这个 bug 开始，这是你唯一能够确定新的代码修复了这个问题的方法，而不是因为不同的环境导致问题看起来消失了。当你应用新的代码时，还希望能够捕捉到所有可能引入的回归或问题。只有你确信问题是由你刚应用的代码引入的，才可以这样做。

设置好环境并确认代码的当前状态后，你现在可以签出一份需要查看的代码的副本。

8.5 应用提议更改

在第 2 章中，你了解过几个不同的 Git 访问控制模型。你的项目设置或许有所不同，提议评审分支可能位于主项目仓库中（8.5.1 节）中，也可能位于一个派生的项目仓库的副本中（8.5.2 节）。由于初始设置的指令不同，因此请跳到与你相关的部分。

8.5.1 共享仓库的设置

如果你在一个共享仓库中工作，那么你的设置过程非常简单。只要更新你的本地分支列表即可，如下所示。

```
$ git fetch
```

如果你拥有超过一个远程，那么或许需要显式指定你想要更新的远程的名称。假定你想要更新的远程的名称为 *origin*，那么命令会如下所示。

```
$ git fetch origin
```

如果你使用自动化构建环境，那么或许需要显式拉取你想要评审的分支（如果你在本地没有远程仓库的完整历史记录）。使用你的远程名称替换 *origin*，并用你想要评审的分支名称替换 *61524-broken-link*，如下所示。

```
$ git fetch origin 61524-broken-link:61524-broken-link
```

第三个参数 `61524-broken-link:61524-broken-link` 是一个引用规格（refspec），将远程分支的名称映射到一个本地分支的名称（`[remote_branch_name]:[local_branch_name]`）。根据约定，远程和本地的分支名称相同，以便记忆，但它让分支名重复了两遍，确实让命令

看上去更加复杂。

现在，你可以继续学习 8.5.3 节。

8.5.2　派生仓库的设置

有两种方法可以处理派生仓库的场景。第一种方法是克隆一份包含提议分支的新的远程仓库副本。如果我们正在评审过程中，并且不负责将提议的更改合并回主项目仓库，那么这种方法是合适的。第二种方法是在我们自己的本地仓库中添加一个新的远程仓库，并将更改拉取到我们自己的仓库中的一个新的分支上。第二种方法还允许我们将通过评审的工作合并回主项目仓库。你应该选择适用于你的情况的方法。如果你还不确定，选择第二种方法，并在你的本地仓库中添加远程仓库的引用。

在这两种方法中，我们需要知道远程仓库的 URL，其中包含你想要评审的更改。它的格式可能是或 git@example.com:username/project.git。一旦你获得了远程 URL，就可以继续了。

如果你在使用创建一份新的克隆的第一种方法，请离开你自己的项目仓库副本，也许会到达你的桌面文件夹，然后使用下面的命令来创建一份你想要评审的仓库的克隆。

```
$ git clone https://example.com/<username>/<project>.git
```

回到你刚克隆的新仓库，如下所示。

```
$ cd project
```

现在，你可以继续阅读 8.5.3 节。

如果你在使用第二种方法（在你自己的项目仓库副本中添加一份远程仓库），那么将需要从你的项目仓库中开始。现在，在命令行中前往那个目录。

一旦来到你的项目目录，请立即为包含待评审分支的派生仓库添加一个新的远程仓库。使用你正在评审的工作的作者的用户名，作为远程的名称。例如，如果你在评审 Donna 的工作，而她的仓库位于 https://example.com/donna/likesgin，命令将会如下所示。

```
$ git add remote donna https://example.com/donna/likesgin
```

既然你已经有了一个新的远程仓库连接，请更新对你可见的分支列表，如下所示。

```
$ get fetch donna
```

现在，你可以继续阅读 8.5.3 节。

8.5.3　签出提议分支

现在，你应该位于一个项目仓库中，其中包含了你需要评审的分支。下一步是签出你需要的分支的一份副本。

列出仓库中的所有分支，如下所示。

```
$ git branch --all
```

此命令将会返回一个分支列表。看上去如下所示。

```
* master
  remotes/origin/master
  remotes/origin/HEAD -> origin/master
  remotes/origin/61524-broken-link
```

我们需要评审的代码位于这个列表的最后一个分支中。如果你添加了一个额外的远程来下载你想要评审的分支，*origin* 这个词可能会是 *donna* 之类的。在下面的指令中，用你分配给包含评审分支的远程的别名，替换 *origin* 这个词即可。

```
$ git checkout --track origin/61524-broken-link
```

现在，我们在本地分支中拥有了一份自己的提议更改的副本。这个新的本地分支副本将会被命名为 *61524-broken-link*。通过添加 `--track` 参数，我们在切换到新的分支时使用显式的连接。这意味着如果我们需要运行 `push` 命令来上传我们的更改，Git 将会知道我们想要将我们的更改上传到哪一个仓库。

现在，我们可以开始我们的评审。

8.6 评审提议的更改

首先，让我们使用 `log` 命令看一看这个分支的提交历史记录，如下所示。

```
$ git log master..
```

这个命令给我们提供了对比分支中所有不同提交的完整日志说明（从最近一次提交开始）。如果没有描述性的提交说明，则将工作退回开发者，要求她添加提交说明。第 8 章介绍了如何编写良好的提交说明，第 6 章介绍了如何重塑历史记录（包括使用交互式变基，在之前的提交后添加新的提交说明）。

为了获得一个大概、但更完整的历史记录，使用图形化的 `log` 命令来验证当前的分支。通过使用 `--graph` 参数，你将会感受到这个分支是如何融入项目最近的历史记录中的，如下所示。

```
$ git log --oneline --graph
```

最后，使用 `diff` 命令。这个命令展示了你的仓库中的两个节点之间的差异。这些节点可以是提交对象、分支顶端和暂存区。我们想要比较当前的工作与分支即将合并“到”的地方，一般来说这个地方是 *master* 分支。

```
$ git diff master
```

当你运行这个命令来输出差异时，信息将会以补丁文件的形式展示。补丁文件难以阅读。你看到的行都以 + 和 - 开头。这些行是分别被添加或移除的行。你可以使用上下键来滚动查看这些更改。当你审查完补丁之后，按 q 键退出。如果你想要看到补丁中更改的更简略的对比，那么可以考虑只列出文件，然后每次查看一个更改的文件，如下所示。

```
$ git diff master --stat
$ git diff master filename
```

让我们看一看一个补丁文件的格式，如下所示。

```
diff --git a/jokes.txt b/jokes.txt
index a3aa100..a660181 100644
        --- a/jokes.txt
        +++ b/jokes.txt
@@ -4,5 +4,5 @@ an investigator.
 The Past, The Present and The Future walked into a bar.
 It was tense.

-What did one hat say to another's
-You stay here, I'll go on a head!
+What's the difference between a poorly dressed man on a tricycle and a
well dressed man on a bicycle?
+Attire.
```

前五行告诉我们，我们正在查看两个文件中的差异，行号表示文件从哪里开始出现不同。有一些行出现在更改前，提示上下文。这些行前都有一个空格的缩进。更改的代码行随后以前缀 -（移除的行）或 +（添加的行）标识。

你还可以通过启动 Git 仓库浏览器来查看我们目前看过的相同信息，以获得更佳的可视化摘要。我使用 gitk，通过 brew 安装的 Git 版本中自带这个程序（但 Apple 提供的版本没有）。任何仓库浏览器都足够使用了，Git 网站上提供了很多可视化的客户端（https://git-scm.com/downloads/guis），如下所示。

```
$ gitk
```

当你运行 gitk 命令时，命令行中将会启动一个图形化工具。单击每个提交来获得更详细的信息。很多工单系统还会允许你并排查看合并提议中的更改。即使你像我一样喜欢命令行，我仍然强烈推荐你使用额外的图形化工具来对比更改。在 OS X 上，我喜欢 Kaleidoscope 应用（http://www.kaleidoscopeapp.com/），因为它还允许我找出图片中的差异，而不只是代码差异。

现在，你已经仔细看过了代码，记下你对以下问题的答案。

- 代码是否符合项目确立的编码规范？
- 代码是否控制在工单中说明的范围内？
- 代码是否以最高效的方式，遵循行业最佳实践？
- 代码是否根据所有内部经验和教训，以最佳的方式实现？将你的偏好、样式差异与代码中真正的问题分开很重要。

了解有哪些代码更改之后，你应该继续将这些更改应用到你的本地环境。换句话说，用合适的方式渲染代码。假设这是一个网站，现在是启动你的浏览器并查看提议的更改的时候了。它看起来怎么样？你的解决方案与程序员的设想是否一致？如果看上去不正确，根据你正在评审的更改类型，你是否需要清除缓存，或许需要重新构建 Sass 的输出来更新项目的 CSS ？

现在，是时候用你的任何测试集来测试代码了，比如以下测试。

- 代码是否引入了任何回归？
- 新代码的性能是否比得上旧代码？下载和页面渲染时间是否还符合你的项目性能预期？

- 所有单词是否拼写正确,以及是否遵循任何品牌特定的指南(例如,标题和句子的大小写)?

根据这个特定代码更改的原始问题的性质，可能还有其他明显的问题需要作为代码评审的一部分指出来。理想情况下，你的团队将会制定自己的评审事项清单。

8.7 准备你的反馈

尽你所能创建最全面的列表，包含你能在代码中找到的所有错误和正确的地方。在评审过程中收到某人喋喋不休的反馈十分恼人，因此我们会尽可能避免“还有一件事情”这种情况出现。

假设你现在有一大堆反馈。虽然我不太相信，但也可能你没有任何反馈。发布你的内部评论，使用便于队友使用的方式组织你的评论。对于到目前为止你搜集的所有反馈，可以将它们分成以下几类。

- 代码错误
  它无法编译，引入了回归，没有通过测试集，或者某种程度上明显出错了。这些问题绝对需要进行修复。
- 代码没有遵守最佳实践
  你有你自己的约定，Web 行业有自己的规范。这些修补是十分重要的，但开发者不一定能意识到其中的一些细节。
- 代码的写法与你的预期不符
  你是一个久经考验的开发者，但在开始实践之前，你也不能证明你是正确的。

8.8 提交你的评估结果

根据这个新的分类，你即将参与以退为进的编码。如果问题只是一个拼写错误，并且属于前两个分类，那么将它修复即可。通过减少在开发者和评审者之间代码需要来回传送的次数，你可以提高团队的效率。显而易见的拼写错误并不一定要回到最初的作者那里，不是么？当然，你的队友可能会有一些尴尬，但他们会感谢你节省了他们的时间。希望下次他们不会犯这样的错误。但是，如果这是第四次或者第五次了，不要修复这个问题。你的时间同样是宝贵的，你的队友需要在给你评审之前检查他们自己的代码。

如果你正在纠结的更改属于第三个分类：立即停止。不要触碰代码。相反，在这个问题第一次被发现的地方更新这个工单，并找出你的队友为什么使用了那种方法。询问“你为什么要在这里使用这个函数”或许会引出一段非常有趣的对话，这段对话中有关于使用这种方法的好处。它或许还能向原作者展示这种方法的缺点。通过开始一段对话，评审可以提升整个团队中的知识水平。通过开始这段对话，你还可能会发现，或许你的解决问题之道不是唯一可行的方案。

如果你“需要”对代码做任何修改，这些修改应该足够小而无足轻重。你不应该在同行评审环节作出任何实质性的修改。完成这些微小的编辑，然后使用如下方法，将这些更改添加到你的本地仓库。

```
$ git add --all
$ git commit -m "Correcting <list problem> identified in peer review."
```

你可以简述你的更改，因为它应该无足轻重。此时，你应该将评审后的代码推送回服务器，供原作者测试和评审。假设你将这个分支设为跟踪分支，应该只要运行以下这个命令即可。

```
$ git push
```

根据你的评审更新 issue 队列。也许代码需要继续修改，也许它写得足够好，现在是时候将这个 issue 队列关闭了。

重复本节中的步骤，直到提议的更改已经完成，并且准备好合并到主分支中。

8.9　完成评审

到目前为止，我们一直在比较工单分支和仓库中的 *master* 分支。评审过程的最后一步将会是把工单分支合并到仓库中指定的主分支（*master*），并清理相应的工单分支。

让我们从更新我们的 *master* 分支开始，确保我们可以在合并后发布我们的更改，如下所示。

```
$ git checkout master
$ git pull --rebase=preserve origin master
```

深呼吸，将你的工单分支合并回你项目仓库中的主分支。正如其名，这个命令将会在你的仓库历史记录中创建一个新的提交，用于在必要时使用 `revert` 命令取消合并某个分支的公共副本，如下所示。

```
$ git merge --no-ff 61524-broken-link
```

这个合并要么成功，要么失败。如果合并失败了，代码的原作者通常能够更好地弄清如何修复合并错误，你可能需要请求他们来帮你解决这些冲突。第 6 章中介绍了处理合并错误的建议。

你的团队在使用哪种分支策略？

使用流水线型的主线分支策略的团队应该确保他们使用 `rebase` 命令同步他们的工作分支 *61524-broken-link* 和目标分支 *master*。在签出目标分支后，新的工作应该使用 `--ff-only` 而不是 `--no-ff` 来合并。它将忽略合并提交，移除工单分支的踪迹，并留下清晰的图形化历史记录。看看你的软件正在使用哪种分支策略，并由此决定你应该使用那种方法来合并你的工作。

一旦分支合并后，你就可以将 *master* 分支上传至中心仓库，以共享修订后的 *master* 分支，如下所示。

```
$ git push
```

一旦新的提交顺利地整合到 *master* 分支后，你就可以删除本地仓库和中心仓库中旧的工单分支的副本。此时只剩下以下这些基本清理工作。

```
$ git branch --delete 61524-broken-link
$ git push origin --delete 61524-broken-link
```

8.10　小结

同行评审环节可以帮助你的团队。我发现它在让想法变成代码之前促进了交流。作为一个额外的好处，它通常鼓励开发者开始寻找自动化测试流程的方法，以提高评审效率。是的，它会花费更多时间，但如果你能看到它带来的好处，我相信这些时间是值得的。

第 9 章

寻找并修复bug

即使有最好的评审环节，有时也会将 bug 引入生产环境。或许这个 bug 是由于合并失误或者测试没有覆盖的场景引入的。不管问题的根源是什么，Git 将会帮助你找出在何时由谁引入了这段讨厌的代码。这将使你理解出现在你的系统中的这段代码的上下文，并在也许你不熟悉的那块代码库中，告诉你能帮你解决问题的最佳人选。

主要有两种方法可以应用你的调查技巧：使用现有代码来定位问题，和使用代码历史记录来定位问题。当你同时使用这两种技巧时，将会是最高效的。例如，在我调试代码的时候，几乎总是从阅读代码开始。这是我从做过的各种 Web 前端开发中学到的，如使用 Firebug 来调试一个网页，找到有问题的 CSS。它绝对不是调试代码的唯一方式，而且对于很多项目来说，这都不是一个可行的策略。

在本章中，你将会学到以下内容。

- 使用 `stash` 保存当前工作的进度，以签出到另一个分支
- 使用 `blame` 寻找文件的历史版本
- 使用 `bisect` 寻找最近一个正确的提交

在本章结束前，你还将更好地理解你现在在 Git 中储存历史记录的方式将会影响到你以后从错误中恢复的方式。你将会意识到一个良好的提交说明是多么重要，同时看到变基工作流如何帮助你创建更容易被 `bisect` 理解的历史记录。本章不会包括如何撤销找到的错误的说明，因为这些内容在第 6 章中已经提到。

喜欢通过视频教程学习的读者可以参考本书附带的系列视频，Collaborating with Git（http://shop.oreilly.com/product/0636920034872.do）。

9.1 使用stash进行紧急的bug修复

在第 6 章中，你学到了如何调整你的提交说明，但为了防止意外，事实上暂时搁置你的工作可能会更合适。这可以通过 `stash`（保存进度）命令完成。这个命令允许你临时搁置你进行了一半、以后某个时候想要继续的工作。

现实世界中 Git 的应用

我最喜欢的一个 Git 相关的笑话是我的朋友 Jeff Eaton 在 DrupalCon Prague 说的。他十分应景地作出了这个评论："像 git stash 那样可以暂时不用考虑是否合乎道德。"我希望我还记得当时的情景（是恐怖电影？还是在啤酒吧）[1],但这个小笑话确实打动了我。

从代码的角度来看，`stash` 允许你避免需要以后重做的无用提交。这些无用的提交通常在你解决一个问题，但需要临时切换到一个不同的分支时出现，而你只有在工作目录干净时才可以切换分支。与分支或单个提交不同，stash 不能被共享，它只存在于你的本地仓库中。

为了新建一个包含你当前工作目录中更改的 stash，你需要使用 `stash` 命令。如果你喜欢明确，可以加上 `save` 参数。不过，这个参数是默认的，如果你想少敲几个键，就不需要加上了，如下所示。

```
$ git stash save

Saved working directory and index state WIP on master: \
  d7fe997 [9387] Adding test: check user exists
HEAD is now at d7fe997 [9387] Adding test: check user exists
```

你将会注意到这个命令只会保存 Git 已知的文件的进度。如果你有尚未提交过的新文件，这些文件将不会被加入 stash，而其他更改会进入一个 stash，使你无法在清理完所有未跟踪的更改前切换到一个不同的分支。为了加入未跟踪文件，请添加 `--include-untracked` 参数，如下所示。

```
$ git stash save --include-untracked
```

或者，如果你想要丢弃这些文件，而不是将它们放到你的 stash 中，可以运行以下命令。

```
$ git stash save
$ git clean -d
```

每次你在不干净的工作目录中使用 `stash` 命令时，一个新的 stash 将会创建。通过添加 `list` 参数，你可以看到一列你保存的 stash，如下所示。

```
$ git stash list

stash@{0}: WIP on master: d7fe997 [9387] Adding test: check user exists
```

如果你只是想花几分钟回顾一个 stash，这没什么问题。你短期的记忆完全能够记住几分钟

注 1：就像看恐怖电影时，我们或许会觉得有人被杀死很有趣，但这仅限于看电影的时候。——译者注

前发生的事情，但如果你想要记住更长时间，并且有更多需要记住的回忆，你会很难想起每个 stash 中有什么。

使用 `show` 命令来查看一个 stash 中的内容。选中的 stash 补丁中的元信息和工作目录中保存进度的更改将会展现，如下所示。

```
$ git show stash@{0}
```

如果你认为你无法通过查看代码想起做过的事，那么可以编写一段精简的描述替代提交说明，包括你在保存工作目录的进度时正在进行的工作。

添加一段描述

如果你想要添加一段描述，那么将需要显式添加 `save` 参数。

Git 允许你储存多步 stash，因此如果你正在解决一个庞大的问题，这将会尤其有用，最终会在同一个分支中创建多步 stash，如下所示。

```
$ git stash save --include-untracked "terse description of the stashed work"
```

现在，如果你再次检查 stash 列表，将会看到你之前的 stash 和新的 stash，如下所示。

```
$ git stash list

stash@{0}: On master: terse description of the stashed work
stash@{1}: WIP on master: d79e997 Revert "Merge branch 'video-lessons' ...
```

最新的 stash 将会出现在列表顶端。注意 stash 的引用编号在你创建更多 stash 时是如何变化的，这个赋值是可变的，而不是一个持久的引用编号。如果你在同一个分支中创建了多个提交，这会有些令人困惑，但如果你给每个 stash 一个精简的描述，那么在继续工作时就更容易想起你想要应用的 stash，以及哪些 stash 已经过时而需要被删除。

保存进度的工作可以应用于任何分支

如果你意识到你在错误的分支上工作但还没有提交，那么也可以使用这个命令。你可以保存你的工作进度，切换分支，然后重新应用 stash 中的工作。

一旦你准备好继续工作，就需要确定你想要使用哪个 stash，然后应用它，如下所示。

```
$ git stash list
$ git stash apply stash@{0}
```

如果你使用 `apply` 命令，这个 stash 将会持久化。如果你的 stash 开始堆积，这会有些困惑。要想移除某个 stash，可以使用 `drop` 命令来删除它，如下所示。

```
$ git stash drop stash@{0}
```

如果你知道自己就是喜欢屯东西，并且认为自己不擅长清理旧的 stash，那么应该使用单个

pop 命令来 apply（应用）和 drop（丢弃）这个 stash。假设你只有一个 stash，那么命令将如下所示。

```
$ git stash pop
```

你还可以使用与 apply 和 drop 相同的结构来 pop（弹出）特定的 stash，如下所示。

```
$ git stash pop stash@{0}
```

有疑问时，Git 假设你想要最新的一个 stash

如果你只保存了一个 stash，则不需要列出所有你想要的 stash。如果你省略 stash 的名称，而 stash 不止一个，那么 Git 将会使用最新的一个 stash（列表中最顶部的那个，它将被命名为 stash@{0}）。

现在，你应该能够使用 stash 命令来临时搁置你的工作。尽管你随时都可以保存你的工作进度，但应该在真的被打断了的时候才使用这个命令。如果你完成了一份完整的工作，请使用 commit 命令。如果你决定以后添加更多工作，那么永远可以选择 rebase 你的分支，合并你之前做出的提交。

9.2　比较历史记录的研究

将出错的代码和另一份代码相比较，是用于查找代码为什么无法工作最基本的方法之一。你可以使用相对历史记录来轻松地做到这一点。与阅读某个特定分支的日志相比，你可以将一个分支与另一个分支，或另一个时间点相比较。

大多数命令以前已经出现过，但这次，带着具体的问题来看它们。考虑图 9-1 中的提交历史记录图。有两个分支拥有共同的历史记录：其中一个包含一个已知的 bug，而另一个确信是正确的。与包含正确代码的分支相比，包含错误代码的分支有四个不同的提交。正确的分支只有两个新的提交，不包括在错误的分支中。

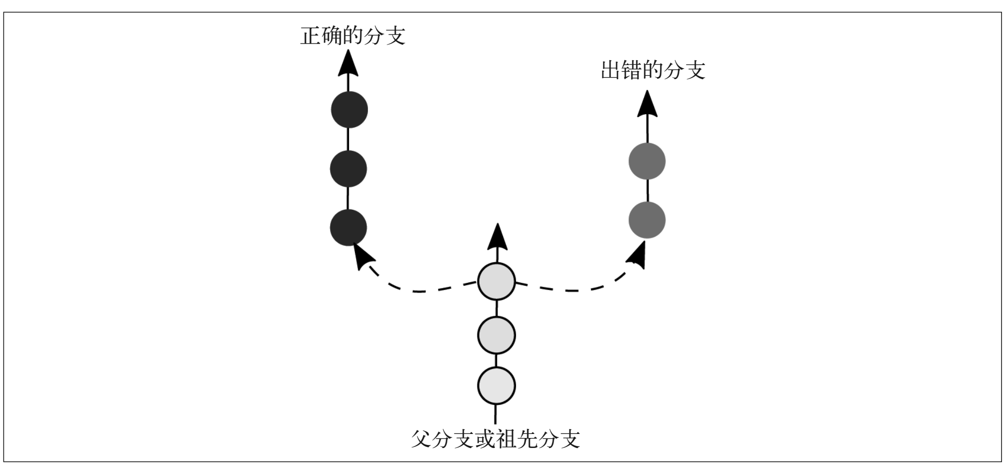


图 9-1：具有共同祖先的两个分支分叉出不同数量的分支

需要一个样例仓库来练习？

如果你想要尝试接下来的练习，请从“Git 团队协作”网站（http://gitforteams.com/）下载一份仓库的副本。这个仓库设置了必要的分支，这样你就不需要自己复制这个场景。

使用 `log` 命令，你可以隔离历史记录的多个碎片。在笔记本上画一张图，在每个命令表示的提交上画一个圆圈。你也可以试着使用 `diff` 取代这些命令中的 `log`，来获得不同的输出。

在当前分支上，我使用下面这个命令来查看除了最近一次提交之外的所有工作。

```
$ git log HEAD^
```

在当前分支上，我使用下面这个命令来查看除了最近三次提交之外的所有工作。

```
$ git log HEAD~3
```

你可以站在不同的参考点比较这些提交。你站在一幢高楼的窗边，俯视楼下的街道。你可以看到其他矮小一些的建筑的屋顶。现在，想象你站在街上仰视这幢高楼。你可以看到人们坐在咖啡馆的遮阳伞下。在 Git 的上下文中，这意味着你可以使用任意分支来作为参考点，如下所示。

```
$ git log since_last_merge_to..what's_been_added_here --oneline
```

例如，我使用以下这个命令来查看包含在正确分支中而非错误分支中的提交。

```
$ git log working..broken
```

相反呢？如何查看包含在错误分支中而不是正确分支中的提交？可以像下面这样做。

```
$ git log working..broken
```

如果我想查看包含在错误分支中而非正确分支中的代码，我会像下面这样做。

```
$ git difftool working..broken
```

你也可以比较远程分支。不要忘记在比较前使用 `fetch` 下载最新的版本，如下所示。

```
$ git fetch
$ git log working..remote_nickname/broken
```

如果你无法获得足够的信息，那么可以使用带 `-S` 参数的 `log` 来查找提交说明，或提交的更改中被应用（或移除）的特定文本字符串。如果你在提交说明中使用了固定的词汇，使用这种方式查找你的仓库会明显更有帮助。例如，我总是试着在提交说明中加入文件名，或等价的缩写，以便日后进行筛选（当这个文件被添加到本书的仓库中去时，它会得到一个包含 `CH09` 的提交说明），如下所示。

```
$ git log -S foo
```

如果 `-S` 参数让你感到惊喜，那么好消息来了！你还可以使用 `-G` 参数来根据正则表达式查找。

使用这些命令可以帮助你隔离造成问题的文件。一旦你找到了文件名，就可以更仔细地调查这些文件。

9.3 使用blame调查文件历史版本

在团队中协作时，查看谁之前修改过一份文件将会十分有用。文件的修改者是有资格讲解历史更改为什么做出的最好人选，尤其是当提交说明没有给出更多线索时。一般来说，我们使用 `log` 命令来查看仓库演进的历史。但它无法给出一个很好的摘要，关于这些更改是如何拼到一起变成你正在查看的文件的。

`blame` 命令允许你逐行查看一份文件，显示每行的最近更改时间、作者和包含更改的提交（图 9-2）。

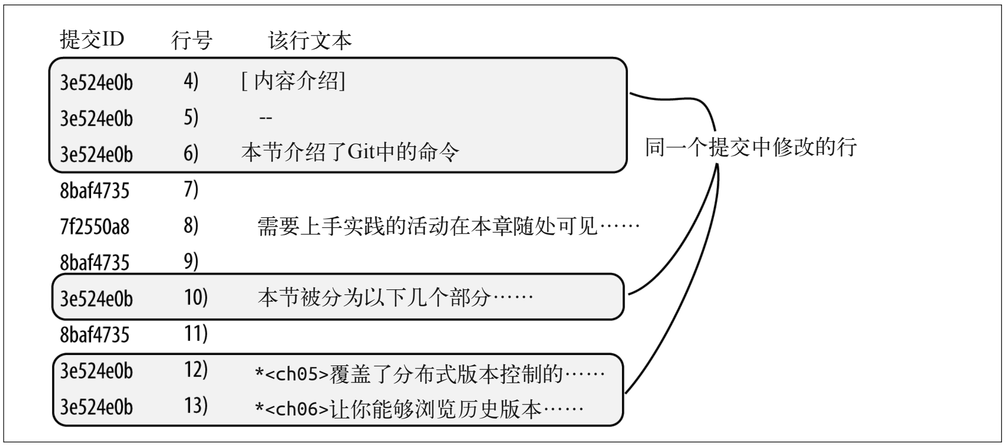


图 9-2：`blame` 允许你通过提交 ID 和作者，列出每行加入文件的时间

为了检验 README.md 文件，请使用例 9-1 中显示的 `blame` 命令，如下所示。

例 9-1　`blame` 命令的输出

```
$ git blame README.md

3e9dd558 (emmajane          2014-04-23 22:11:40 -0400  1) Git for Teams of One...
^00de359 (Emma Jane         2014-04-23 18:54:03 -0700  2) ====================
^00de359 (Emma Jane         2014-04-23 18:54:03 -0700  3)
3e9dd558 (emmajane          2014-04-23 22:11:40 -0400  4) Supporting files for ...
7874193c (emmajane          2014-06-26 00:37:41 -0400  5) developer work flow for ...
3e9dd558 (emmajane          2014-04-23 22:11:40 -0400  6) version control system, git
3e9dd558 (emmajane          2014-04-23 22:11:40 -0400  7)
00000000 (Not Committed Yet 2015-01-15 21:08:09 +0000  8) Test edit!
00000000 (Not Committed Yet 2015-01-15 21:08:09 +0000  9)
3e9dd558 (emmajane          2014-04-23 22:11:40 -0400 10) ## Contents
3e9dd558 (emmajane          2014-04-23 22:11:40 -0400 11)
5cc35764 (emmajane          2014-06-25 17:45:38 -0400 12) */slides*
3e9dd558 (emmajane          2014-04-23 22:11:40 -0400 13)
```

从左到右，这些列显示了以下各项。

- 提交的散列 ID
- 作者名

- 日期
- 行号
- 这个文件中特定行的内容

在例 9-1 中，你或许已经注意到其中列出了三个作者：Not Committed Yet、emmajane 和 Emma Jane。希望第一个名称能够解释自己：这些更改位于我的工作目录，但尚未被提交。我两个版本的名称是由于我的 Git 设置变动造成了不一致。你可以在附录 C 中读到更多如何自定义称呼的内容。

其中两行以 ^ 开头。这两行自从第一次提交后没有再被修改过。

> **小心！“blame”（责备）这个词可能会诱导你产生负面的想法**
>
> `blame` 命令的命名很糟糕。它瞬间就不必要地制造了对代码的负面看法。我更喜欢用于 Git 的竞争对手 Bazaar 的命令 `annotate`（批注），以及它的别名 `priase`（称赞）（完整地说，Bazaar 对于 `annotate` 还有一个别名 `blame`）。Git 没有 `annotate` 命令，但这个命令的文档说明只是出于兼容性考虑。它不是一个真正的别名，而 `blame` 和 `annotate` 的别名有些许差别。
>
> 最后一个修改某行代码的人通常最有资格解释他们的目的，提前去找他们的麻烦会减少以后他们向你寻求帮助的可能性，同时可以避免增加你需要解决他们未来错误的可能性。检查你在使用这个命令时的态度，看看你能不能从归咎的思维方式转换成简单的批注。

一旦你定位到了文件中看上去有点意思的一行，可以使用提交 ID 以及 `log`、`diff` 和 `show` 命令来继续深究。表 9-1 列出了每个命令能帮你隔离什么。

表9-1：使用`log`、`diff`和`show`的原因

描述	命令
显示某个特定提交的元数据	`log` *commit*
显示某个特定提交中更改的代码	`show` *commit*
显示某个特定提交后更改的代码	`diff` *commit*

从使用 `log` 命令查看提交说明开始，如下所示。

```
$ git show <commit>
```

如果提交说明编写得很好，那么其中应该包含关于这个特定提交中为什么做出了这些更改的说明。如果详细的提交说明中包含一个指向项目管理系统中工单号的引用，你甚至可以阅读这个更改的讨论，让你深入了解开发者在创建这个修复时是怎么想的。在跟踪系统中，你还可以看到其他参与的开发者，以及这个特定更改的评审团队的所有成员。

要想看到相同数量、但在某个节点之后所有提交的详细信息，可以使用以下命令。

```
$ git log --patch <commit>
```

在这个上下文中，`--patch` 参数向你展示每个提交之间的更改（而 `diff` 命令是向你展示引

用提交和工作目录中的文件之间差异的）。

blame 只能告诉你可见的内容

blame 不是万能的。如果 bug 并非是由你正在查看的版本的文件中的某行引入的，blame 将不会告诉你谁最近编辑了这份文件。因此，这是一个好用的工具，但不是魔法。

组合使用 blame、log 和 diff 命令，你现在应该可以在单个文件的完整历史记录中和同时发生的其他更改的上下文中评审这个文件的历史记录。使用提交说明，你或许还能够追踪更改发生的原因。通过一些细致的调查，你可以在询问代码作者时展开一段富有成效的对话，而不是一段神探可伦坡风格的审问。

9.4 使用bisect重演历史

如果你不知道问题在哪个文件中，通常很难找到 bug 是什么时候进入你的代码中的。如果你正在查看的错误消息被输出到屏幕上，相对来说查找代码库中的文件来定位正确的文件就更容易了。在错误发生时，有时错误消息会包含文件名和行号。在任何情况下，你都可以使用 diff、log 和 blame 命令来得到对错误来源的更好理解。有时有问题的代码并没有在错误消息中留下足够的线索来使用这些工具。现在，该 bisect 来大显身手了！

bisect 对以往的提交执行二分查找，来帮助你找到代码从一个已知正确的状态到一个已知错误的状态之间的提交。与平时的签出提交不同，bisect 继续游荡于你的历史记录中（以一种非常合理的方式），直到你有了足够的线索来辨认是哪个提交引入了错误的代码。这有点像重演开发者在代码库中的工作历史。在 bisect（二分查找）过程的每个节点上，你可以启动这个产品（编译代码，在浏览器中打开，在你的手机上安装这个应用，进行任何对代码库来说合适的事）并确定历史上此时的代码是正确的还是错误的。一旦你找到出错的时间点，就可以在那个时间点上修复历史记录。就像电影《回到未来》中那样，Git 就是你的 DeLorean。

要想开始，你需要来到仓库最外层的目录。隐藏的 .git 文件夹就位于这个文件夹中。开始 bisect 过程，并告诉 Git 一个代码正确的提交 ID 和一个代码错误的提交 ID（例 9-2）。

例 9-2　标记正确和错误的提交以进行二分查找

```
$ git bisect start
$ git bisect good <commit-id>
$ git bisect bad <commit-id
```

现在，Git 将会继续每次签出一系列提交，寻找代码从错误变回正确的那个提交，如下所示。

```
$ git bisect start
$ git bisect bad c04f374
$ git bisect good 93b64fc

Bisecting: 10 revisions left to test after this (roughly 4 steps)
[0075f7eda67326f1746] Merge branch 'video-lessons' into integration_test
```

仓库现在处于分离式 HEAD 状态。此时，你需要确认这段代码是正确的还是错误的，并报告你的发现，如下所示。

```
$ git bisect bad

Bisecting: 5 revisions left to test after this (roughly 3 steps)
[ed8056eb4b2aaf00e6d] Lesson 4: Adding details on using git config

$ git bisect bad

Bisecting: 2 revisions left to test after this (roughly 1 step)
[c88a02babc42bb00a83] Lesson 4: Adding new lesson on configuring Git

$ git bisect good

Bisecting: 0 revisions left to test after this (roughly 1 step)
[f1fa8e7e382f68c0558] Lesson 3: Extended descriptions for cloning a ...

$ git bisect good

ed8056eb4b2aaf00e6d is the first bad commit
commit ed8056eb4b2aaf00e6d9d183f974ed612d6f10e6
Author: emmajane <emma@emmajane.net>
Date:   Sun Sep 7 12:50:58 2014 +0100

    Lesson 4: Adding details on using git config

    Added commands to customize the following:

    - username (or real name, as you prefer)
    - email address
    - enable color helpers within the git messages

    Added a self-study piece on customizing your command prompt to include
    additional color and branch information.

:040000 040000 e927a1263e6e23eb5237a363a20640f62349b27d
31bc6c57d6acd8de214a63a47914b32d6809a866 M      lessons
```

有问题的提交已经找到了。此时，你处在分离式 HEAD 状态，但也知道了你需要回到哪个提交。要带着新的信息回到你的分支顶端，请使用子命令 reset。这个命令还可以在 bisect 过程的任何节点上用于中止查找并回到分支上最近一个提交，如下所示。

```
$ git bisect reset
```

如果你还没有经历过大量编程，二分查找过程可能会像是魔法。（是真的非常酷炫的魔法哦。）如果你想要减少一些神秘感，可以使用子命令 visualize 来展示 bisect 过程的最新状态（图 9-3）。两侧正确和错误的提交将会在你刚配置过的 gitk 的图形界面中进行标记。

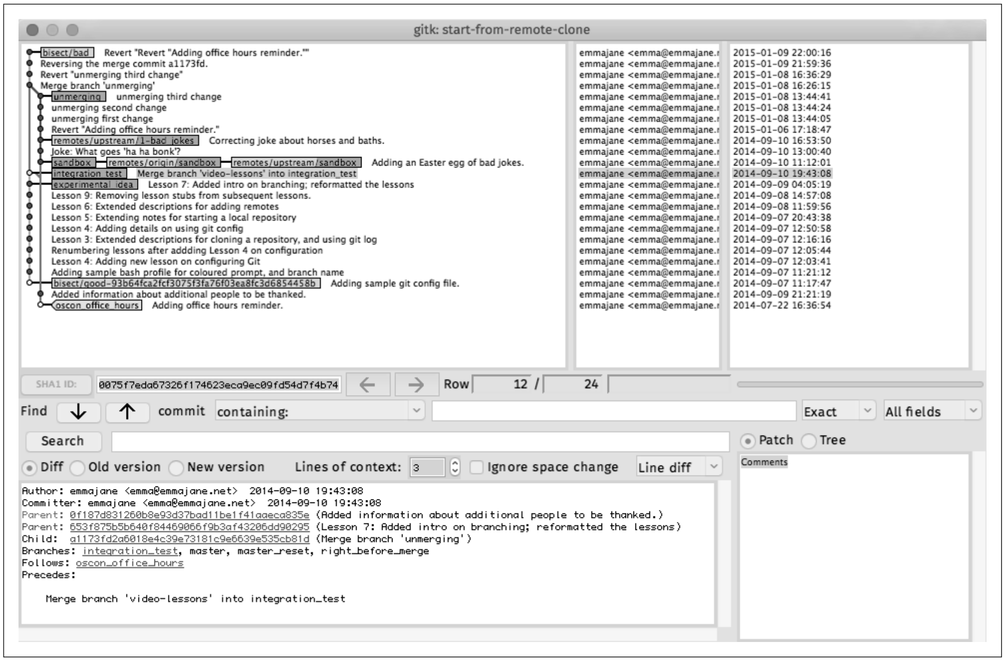


图 9-3：运行 `git bisect visualize` 展示 bisect 过程的最新状态

> **bisect 假定不好的事情已经发生**
>
> 我们假设当前的工作是有问题的。因此，你不能回到过去然后寻找问题被修复的时候，而需要回到过去并找到出错的地方。如果你试着寻找什么时候加入了一个修复，尽管这是可能的，但它看上去会非常令人困惑。你只需要记得对调正确和错误的定义。

9.5 小结

我会愉快地承认我痴迷于犯罪电视剧，因此本章中使用 Git 进行深入调查十分吸引我。在本章中，你接触到了下面这些我放在侦探工具箱中的新命令。

- `stash` 允许你搁置你当前的工作，这样你就可以签出另一个分支了。
- `blame` 允许你查找一份文件的逐行历史记录。
- `bisect` 允许你有条理地查找历史记录，来寻找出错的地方。

这些工具和第 6 章中从错误恢复的信息一起，将会帮助你深入探索你可能会遇到的任何犯罪现场并从中恢复。

第三部分

Git托管平台

本书前两部分介绍了 Git 通用的命令，而不是针对任何一个代码托管平台。在第三部分中，你将学到三个流行的协作平台：GitHub、Bitbucket 和 GitLab。在我参与过的很多项目中，我发现我的工作通常分为三块：在 GitHub 上托管的开源项目，在 Bitbucket 上托管的私有客户工作，以及 GitLab 上的内部自治项目。

并不存在任何正式的限制规定你必须以这种方式使用这些系统。实际上，GitHub 也有企业版本，允许你购买一个“本地部署”的 GitHub 示例；GitLab 也有一个社区版本，提供免费的私有和公开 Git 仓库的托管。

有很多书专门讲述了如何使用这三个平台中的每一个。后续的每一章并不会尝试重复这些工作，而是设计为我最常见到的这些平台的用法的“入门”指南。第 10 章介绍使用 GitHub 托管公开的开源项目，第 11 章介绍使用 Bitbucket 托管私有的闭源项目，而第 12 章介绍使用 GitLab 托管私有的内部仓库。

第 10 章

GitHub上的开源项目

GitHub 拥有 900 多万名用户，是当今世界上最大的代码托管平台。如果你是 Web 开发者，或者正在参与开源软件的开发，那么你很可能最近访问过 GitHub 网站来下载一些代码（如果你没有创建账户并且参与到开发社区）。进行专有代码开发的程序员可能不那么了解 GitHub，但这并不妨碍它是一个代码托管平台，因为 GitHub 同样允许你创建私有仓库（如果你不希望共享你的代码）。

本章的重点将是使用 GItHub 进行公开的项目开发，因为它已经成为大多数新人首先接触的系统。学完本章，你将能够在 GitHub 上完成以下事项。

- 创建新账户
- 创建组织
- 创建新项目
- 接纳新的协作者的贡献
- 接受协作者的拉取请求

到目前为止，你一直使用的仓库样例都托管在 GitLab 上。与 GitLab 不同，GitHub 平台并不是开源软件。GitHub 毫无疑问可以提升你使用 Git 的经验，但 GitHub 自身的一些用途有时候会让你难以分清你什么时候在使用 Git 术语，什么时候在使用 GitHub 术语。

作为 Web 开发者，我可以利用 GitHub 的很多很棒的功能。我曾经使用 GitHub 发布简单的静态网站，甚至是基于 HTML 的幻灯片。使用本书之前已讲述的相同方法，你首先将学习如何在单人团队中使用 GitHub，然后学习使用它的功能来与他人协同。当然，如果你已经在某个团队中工作，那么我鼓励你直接跳到本章中最适合你的那一节。

喜欢通过视频教程学习的读者可以参考本书附带的系列视频，Collaborating with Git（http://shop.oreilly.com/product/0636920034872.do）。

10.1　开始使用GitHub

在本节中，你将学习如何在 GitHub 上创建账户，并在自己的 GitHub 账户中发布仓库。本节的目标是让你熟悉单人团队中的 GitHub 使用，因此一些操作可能在参与大型团队时会感觉更加自然些。

10.1.1　创建账户

在 GitHub 上访问公开仓库不需要账户。如果你希望上传代码，或者参与代码的讨论，那么将需要创建一个账户。幸运的是，设置账户非常简单直接，而公开的仓库是免费的。一个免费账户足够完成本章中包含的所有内容。

第 1 步：创建你的账户

(1) 访问 https://github.com（图 10-1）。
(2) 输入一个未被占用的用户名。GitHub 会告诉你这个名称是否已经被使用。
(3) 输入一个合法的电子邮件地址。
(4) 输入一个安全的密码。
(5) 点击 Sign up for GitHub（注册 GitHub 账户）按钮继续。

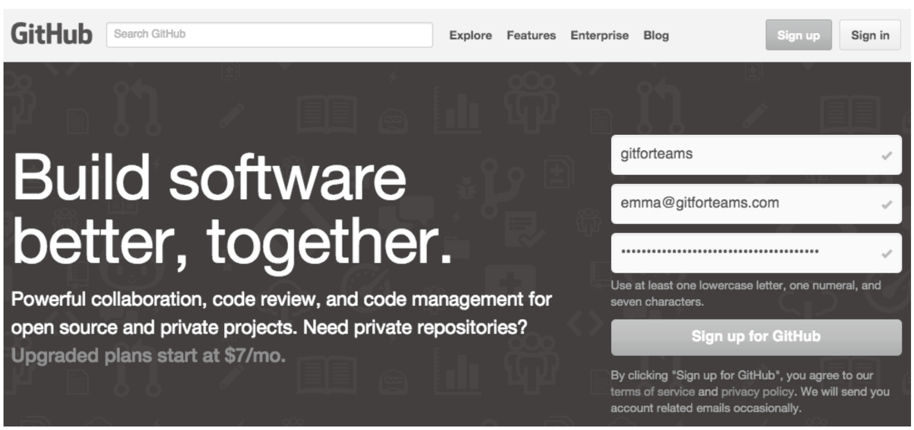


图 10-1：注册 GitHub 账户

在完成用户名是否唯一、电子邮件是否合法和密码是否安全的验证测试后，你将会进入下一个页面。

第 2 步：选择一个付费方案

此时，你可以选择通过付费方案来花钱支持 GitHub。代码托管服务毫无疑问是免费的。默认情况下，GitHub 为你选择免费的方案（图 10-2）。你需要遵循以下步骤。

(1) 确认你希望启用的方案类型。默认情况下，免费方案是被选中的。
(2) 通过点击 Finish sign up（完成注册）完成账户创建过程。

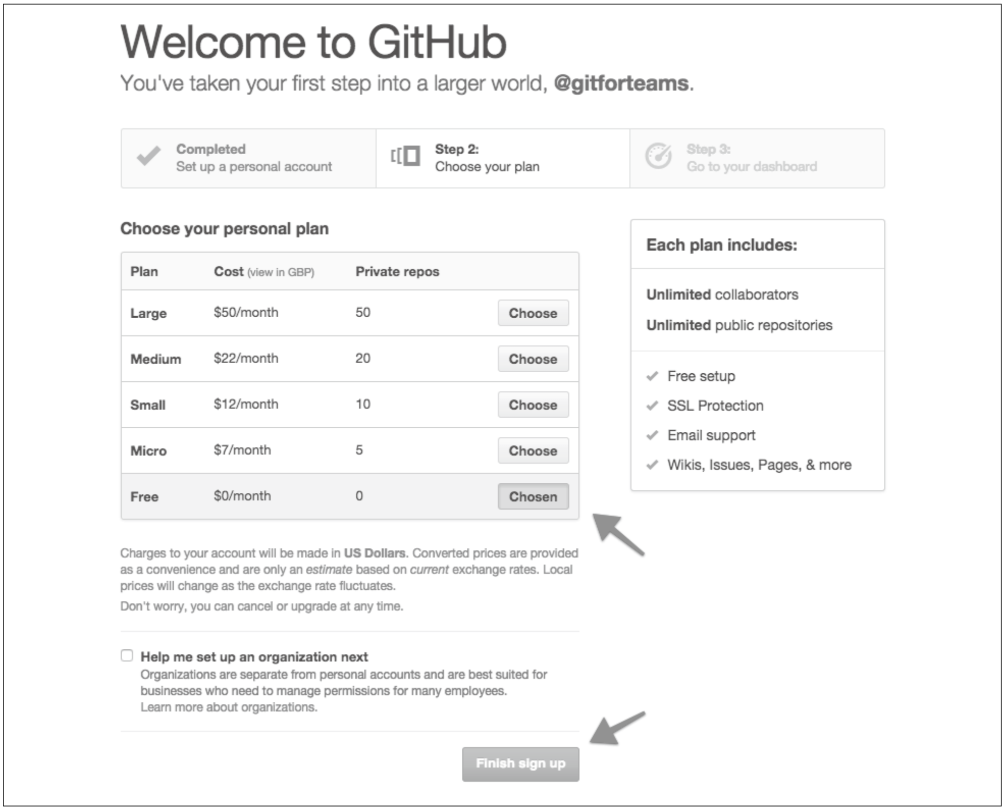


图 10-2：为你的 GitHub 账户选择一个方案

通过付费支持 GitHub 运营

如果你希望帮助 GitHub 继续运营下去，你或许希望在未来某个时候选择付费方案。采取付费方案的好处之一是，你可以创建私有的仓库，只有你添加到项目的开发者才能够访问到。

在创建账户后，你就会收到一封来自 GitHub 的电子邮件，要求你确认你的电子邮件地址。你需要点击此电子邮件中的链接来完成账户的创建过程。如果你不验证电子邮件地址，将无法完成某些操作。

你现在可以使用你的账户来执行一系列的操作，包括创建新的仓库，向你自己和其他人的仓库中贡献代码。

SSH密钥

如果你使用一个非常安全的密码，那么可能会使用密码生成器获得一个长度为 45 位，包括字母、数字和特殊字符的密码。没有人想要重复输入这种密码，但为了授权上传，在将代码上传到 GitHub 上时系统会提示你输入密码。通过使用你的 SSH 密钥来上传代码，你可以避免在每次想要发布代码时重复输入密码。

附录 D 包含了如何创建并获取 SSH 密钥的说明。一旦你将公钥复制到剪贴板后，就可以继续回到 GitHub，操作步骤如下所示。

(1) 访问 https://github.com/settings/ssh。你也可以登录账户，点击右上角的设置齿轮，然后从账户导航栏的选项中点击 SSH Keys（SSH 密钥）。
(2) 在 SSH Keys 摘要页面，点击 Add SSH key（添加 SSH 密钥）。
(3) 可选择是否为你的 SSH 密钥添加一个标题。例如，除了为工作电脑生成的密钥，你可能有一组个人 SSH 密钥。
(4) 将你之前复制的公钥粘贴到 Key（密钥）字段中。
(5) 点击 Add key（添加密钥）按钮。

SSH 密钥必须是唯一的

GitHub 只会允许系统中的密钥被添加一次。如果你已经在别的账户中用过这些密钥，那么在尝试保存这些密钥时会看到一条错误消息。

现在，你可以从你的本地电脑中执行需要认证的操作，但不需要输入你的 GitHub 密码。

10.1.2 创建组织

假设你将会进行一个开源项目，那么现在或许会想要创建一个组织。组织能够管理其中的项目。多个项目可以加入（或被分配给）一个组织。这允许你不用创建另一个 GitHub 账户就可以管理一个项目。组织可以免费创建，所以你同样可以利用它们。

为你的组织命名

一般来说，你将会创建一个与仓库主项目名相同的组织名称。例如，如果你当前仓库中可用的库名为 evilrooster，那么你想要为新的项目账户保留的名称应该也是 evilrooster。一旦新的组织创建后，你可以将仓库的所有权从你的个人账户转移到项目的组织账户。这还将允许你保留仓库的项目历史。

要创建一个组织，请完成以下步骤。

(1) 从顶部的菜单中，点击头像左边的 + 符号。
(2) 点击 New Organization（新建项目）。你将会被重新定向到新组织的设置表单。
(3) 在 Create an organization（创建新组织）的表单中，输入以下信息。

- Organization Name（组织名）：这将会是你项目 URL 的一部分。
- Billing email（账单寄送电子邮件地址）：即使你选择免费方案，这个字段也是必填的。
- Plan（方案）：默认选择开源。

(4) 点击 Create organization（创建组织）以继续。

在下一个页面中，你可以添加组织的团队成员。你自己的账户已经默认添加。为了添加额外的账户，请完成以下步骤。

(1) 在搜索框中，输入你想要添加的成员的名字或用户名。
(2) 点击成员名字右边的 + 符号。
(3) 重复第 (1) 步和第 (2) 步，添加所有成员。
(4) 点击 Finish（完成）来发送邀请。

你的组织已经创建完毕，在你设置组织时，新成员也已添加完毕。

10.1.3 个人仓库

本节将简述把你自己的仓库放到 GitHub 上的过程。你将会使用你的个人账户来创建一个新的仓库，这适用于你一般不希望其他人贡献的项目，因为这些项目就是你的。例如，当我发布会议演讲中使用的 HTML 演示文稿时，通常会将它们推送到 GitHub 仓库来共享。

1. 创建项目

GitHub 上的仓库远不止你在本地电脑上的目录中运行 `init` 命令时得到的 Git 仓库。它提供 issue 跟踪工具，能够将 Markdown 文件转换成网页，还有附带的 Wiki 页面、图表等。然而，GitHub 仍然将这个过程称为创建仓库。

要想开始创建新仓库的过程，找到并点击屏幕右上角的 + 图标，然后选择 New repository（新建仓库，图 10-3）。

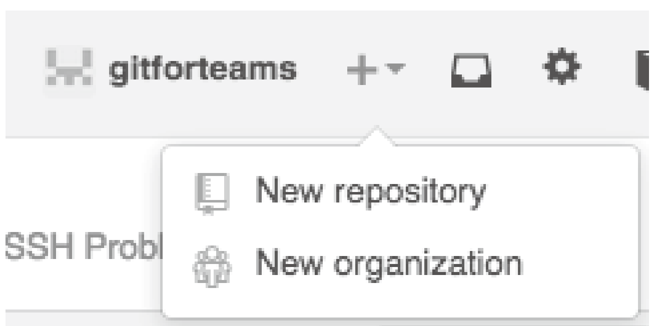


图 10-3：创建一个新仓库

或者，你可以登录并前往 GitHub 首页，然后找到并点击 Create new repository（新建仓库）按钮。

一旦你开始了这个流程，就会来到一个页面，你需要在该页面上填写项目的详细信息（图 10-4）。表 10-1 总结了你需要的信息。

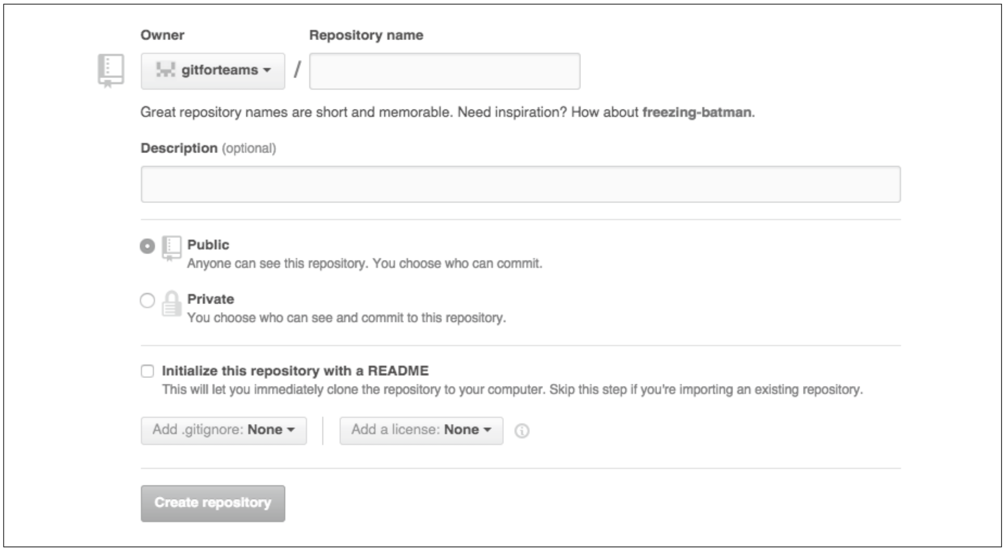


图 10-4：输入新仓库的详细信息

表10-1：用于新建GitHub仓库时需要的详细信息

字段	备注	导入时是否需要?
Repository name（仓库名称）	你的新项目可以在 https://github.com/<username>/<repo-name> 这个 URL 上访问到。请选择短小精炼的名称	是
Description（仓库描述）	在仓库主页顶端，文件列表之上显示的文本	是
Visibility（可见性）	选择 Public（公开，默认选项）或 Private（私有，需要付费账户）	是
Initialize this repository with a README（初始化仓库时添加 README 文件）	添加一个空文件，描述项目的详细信息。这个文件将会在你的仓库首页渲染成 HTML，但可以使用 Markdown 编写	否
Add .gitignore（添加 .gitignore）	很多编程语言会在构建过程中生成不该引入仓库的编译文件。你可以现在生成一个 .gitignore 文件，其中包含你使用的语言中常见的文件扩展	否
Add license（添加许可证）	如果没有许可证文件，你没有给人们下载并使用你的代码的权限。你保留全部版权，不授权别人使用你的工作。理想情况下，你的项目需要包括一份许可证。如果你不确定选择哪一个，选择许可证（http://choosealicense.com/）能够帮助你	可能

如果你已经在本地创建了一个仓库，则可以选择将它上传到这个新项目中；但是，如果你在初始化过程中创建了文件，那么将需要首先下载这些更改，将它们并入你的本地仓库，然后再将它们推送回 GitHub。为了避免额外的这一步，如果我已经在本地拥有了一个仓库，那么将会忽略 README、.gitignore 和许可证文件的创建。

一旦你填写了表 10-1 中每一项的值，找到并点击 Create repository（创建仓库）按钮。你的新仓库将会被创建，而你将会来到摘要页面，该页面上将显示关于下一步该怎么做的建议（图 10-5）。

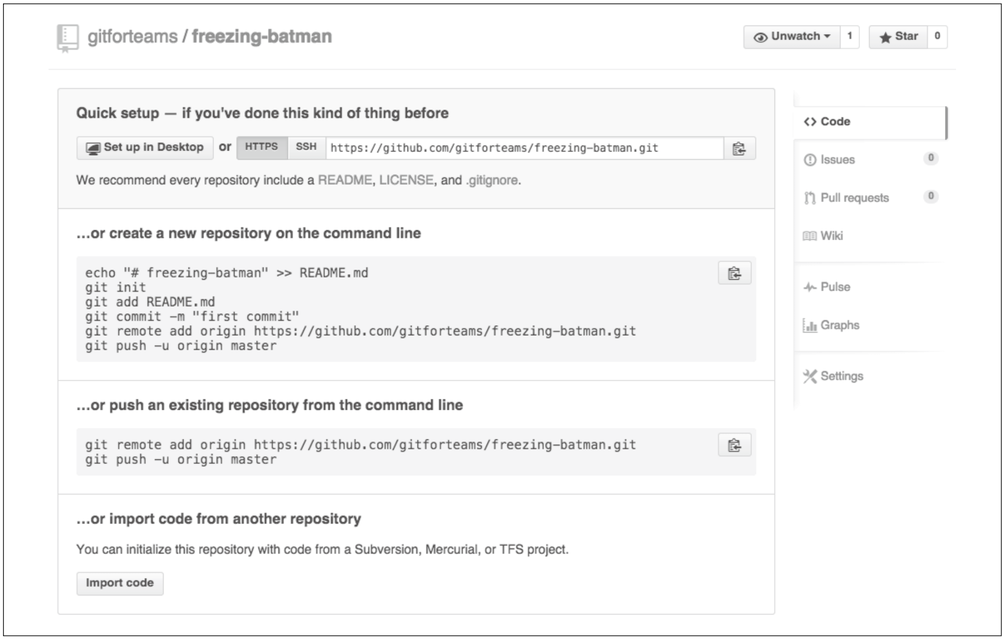


图 10-5：你的新仓库已经可用

因为在仓库的创建过程中没有初始化任何文件，所以你目前只有两个选项：从你的本地电脑上传一个仓库，或者通过一个公开可访问的 URL 导入项目。例如，如果你想要复制本书之前使用的 GitLab 项目，那么可以这么做。我们接下来介绍这些选项。

2. 导入仓库

如果你遵循了本书之前的步骤，那么应该已经创建了一个 GitLab 仓库，克隆了“Git 团队协作”工作坊的文件。你可以轻松地将这个仓库导入 GitHub。只有当你的 GitHub 仓库中没有文件时，你才可以完成这个过程，如下所示。

(1) 访问你的项目主页。
(2) 如果仓库是空的，你可以找到并点击 Import code（导入代码）按钮。点击这个按钮将会打开 GitHub 导入工具。
(3) 输入你想要输入的仓库 URL。它必须是一个公开项目，但并非必须是一个 Git 仓库。你还可以导入 Subversion 和 Mercurial 仓库。如果你在导入一个 Git 项目，请确保你输入了完整的 URL，包括 .git 扩展名，这个 URL 结构与你想要在本地克隆仓库时使用的相同。图 10-6 显示了一个项目的合法 URL（https://gitlab.com/gitforteams/gitforteams.git）。
(4) 点击 Begin import（开始导入）。导入流程将会开始。
(5) 当导入过程完成后，点击 Continue to repository（进入仓库）。你的文件将会从远程仓库中导入（图 10-7）。

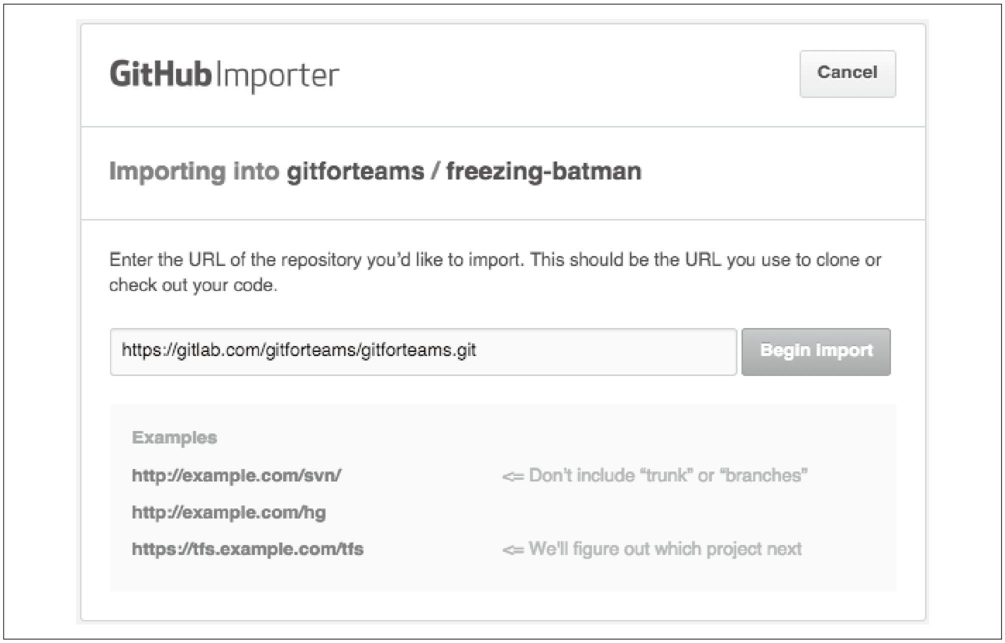


图 10-6：输入 Git 仓库的合法 URL 来将它导入 GitHub

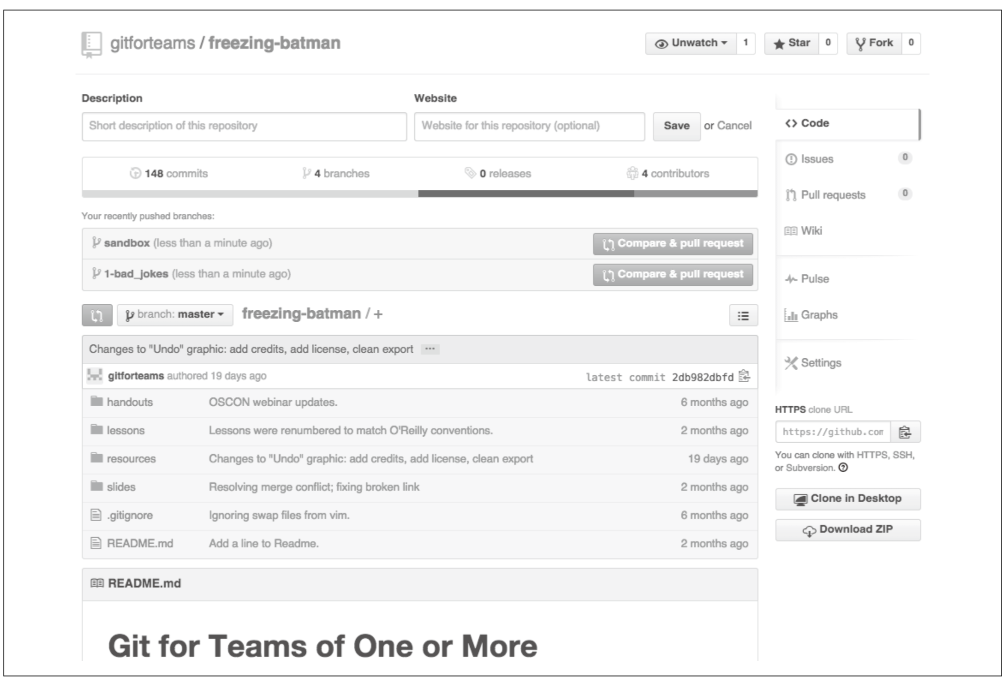


图 10-7：仓库文件和历史已经成功从 GitLab 导入 GitHub

3. 连接本地仓库

在第 7 章中，你学会了如何连接本地仓库和 GitLab 上的一个新的远程仓库。我们会在这里为新的 GitHub 仓库重复这个过程。如果在初始化过程中没有创建文件，GitHub 会在项目主页上提供一组可以直接复制粘贴的命令来完成这些步骤。远程仓库的结构为 https://github.com/<username>/<repo-name>.git。例如，我使用 GitHub 给我的样例名称（glowing-octo-dangerzone）和账户名 gitforteams 创建了一个新仓库。如果我接下来想将我自己的电脑中的一个仓库和这个仓库连接，那么将会完成例 10-1 中概述的步骤。

例 10-1　克隆仓库

```
$ git remote add origin https://github.com/gitforteams/glowing-octo-dangerzone.git
```

一旦你在完成了这些步骤之后访问项目主页，就会看到所有文件已经上传。现在，你可以开始在你的 GitHub 仓库中工作。

4. 将更改发布到你的GitHub仓库

一旦连接了你的本地仓库和 GitHub 仓库，你就可以使用 `push` 命令，将提交的更改上传至任何跟踪分支。为了在 GitHub 上发布一个新的分支，你需要明确告诉 Git 你想将哪个远程作为你的上游分支（例 10-2）。

例 10-2　设置远程仓库中的上游分支

```
$ git push --set-upstream origin master
```

在设置了上游连接之后，你不需要再次添加 `--set-upstream` 命令。如果你希望将你的更改发布到一个以上的远程仓库，则需要继续指定使用的远程。

5. 在网页上进行提交

使用 GitHub 这样的代码托管平台（而不仅仅是命令行）的好处之一是，系统自带了一些微小的改进。例如，GitHub 允许你通过 Web 用户界面编辑任何仓库中的文件。尽管我推荐你不要将它当作你的日常代码编辑器，但如果你只是想修复一次草率的提交中的一个拼写错误，这会非常方便。

为了在 Web 编辑器中进行编辑，你需要完成以下步骤。

(1) 访问你想要编辑的特定文件。这个文件的 URL 将会包含分支名。例如，https://github.com/gitforteams/freezing-batman/blob/master/README.md。
(2) 找到并点击铅笔图标来编辑文件（图 10-8）。或者，在键盘上按下 e。

你将来到一个基于浏览器的文本编辑器（图 10-9）。现在，你可以修改仓库中的这个文件。

在编辑完成后，你可以点击 Preview changes（预览更改）按钮。新增的行中，更改的文本左边有一个绿色的长条（包裹在 HTML 元素 `ins` 中）；被移除的行左侧有一个红色的长条（包裹在 HTML 元素 `del` 中）。在图 10-10 中，第一段长条标记的段落被删除了；第二个段落是新增的。除了颜色和 HTML 元素，似乎没有什么方法可以更好地标识新增和移除的内容差异。

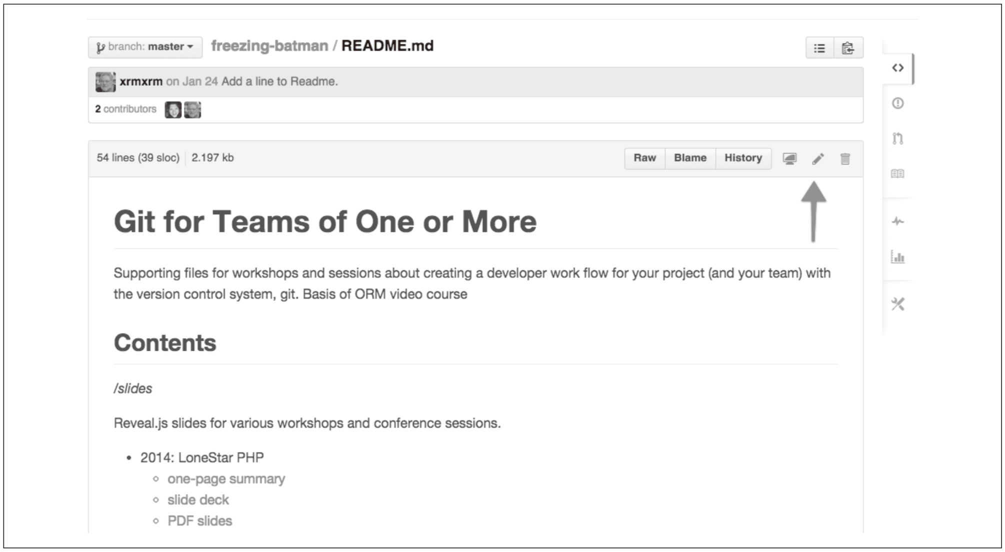


图 10-8：你可以点击铅笔图标来编辑任何文本文件

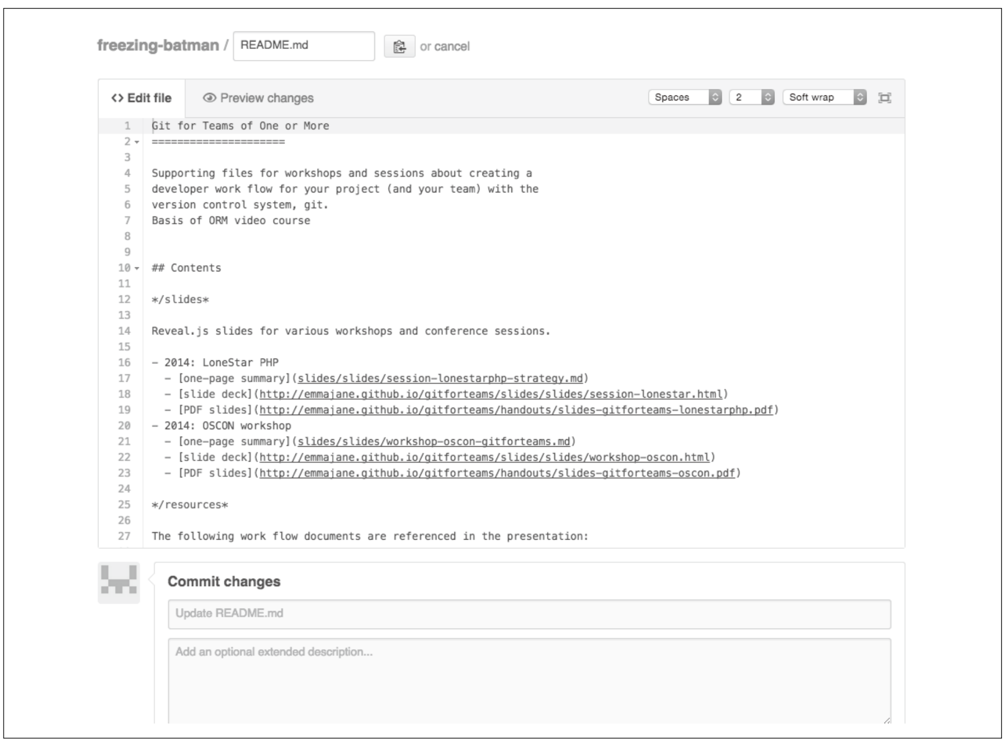


图 10-9：带有可选预览的基于浏览器的文本编辑器

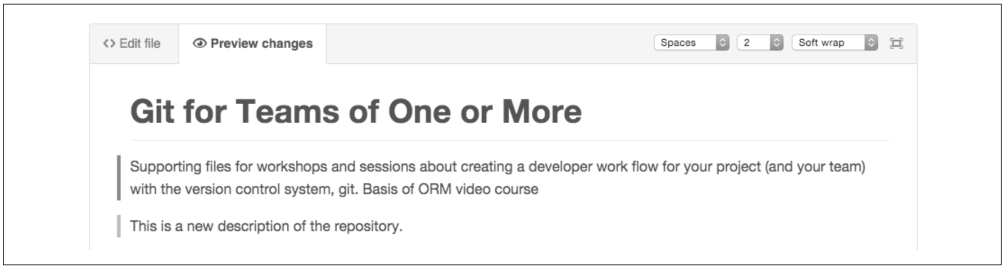


图 10-10：预览显示了哪些行被修改了（第一行被移除，而第二行被添加）

一旦文件中的编辑完成后，你就可以将更改提交回你的仓库（图 10-11）。短提交说明提供了默认值，表明文件的状态更新。您应该对所做编辑进行更具描述性的描述。你也可以添加一段可选的扩展说明。你需要决定是希望将更改提交到当前分支，还是希望为这个更改创建一个拉取请求。默认情况下，GitHub 假设你想要直接将更改提交到仓库中的同一分支。

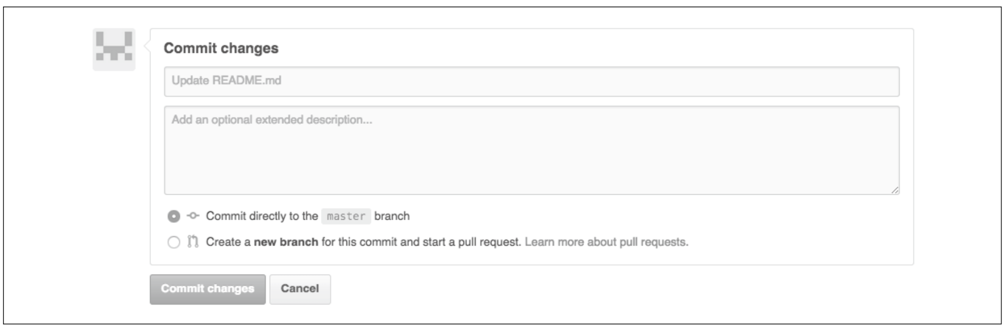


图 10-11：将你的更改提交回仓库

因为你正在你自己的仓库中工作，所以将更改提交到 *master* 分支没什么关系。选择默认值，然后点击 Commit changes（提交更改）。

你为什么可能会想要给自己提交一个拉取请求

如果你是项目唯一的编辑者，则可能不需要为你的更改创建拉取请求。然而，拉取请求并入 *master* 分支时会使用 `--no-ff` 参数。这意味着在图形化的历史记录中，它将会出现在 *master* 分支的直线历史记录外。如果你不介意这个提交只出现在主分支上，那么省略拉取请求这一步是无所谓的。创建和关闭拉取请求的步骤说明将在本章后续部分中介绍。

一旦你将更改提交到仓库之后，就需要更新你的本地仓库来查看这些更改。

6. 更新你的本地仓库

如果你确实使用基于 Web 的编辑器更新了分支，那么你的本地分支将会过时。（不要尝试在你的本地分支上重做相同的编辑，Git 需要同一时刻完全一致的提交来知道两个提交是同一个。）你需要下载这些更改，并在 GitHub 允许将新的更改上传之前，将它们整合到你

的本地仓库。你可以使用下面的顺序来完成这一步。

你应该从你本地的项目仓库目录中开始。接下来，确保你正在使用与远程编辑相同的分支。此分支很可能就是 *master* 分支，如下所示。

```
$ git checkout master
```

接下来，将远程的更改并入你的本地工作。因为这些更改被复制到了同一个分支，而且这些微小的更改不是新的功能，所以我会使用 `--rebase` 而不是 `merge` 选项来并入这些更改。这将保持图形化的历史记录便于阅读，如下所示。

```
$ git pull --rebase=preserve
```

你的本地分支应该已是最新的，准备好进行新的工作。

10.2 使用GitHub上的公开仓库

在为项目工作时，你可以选择下载一份文件的压缩包，或者维持一个远程仓库的连接，在新的更改存在时下载，并将你可能的更改贡献回项目。在本节中，你将学到如何使用 GitHub 上的项目，而不是提交贡献。后者将在下一节中介绍。

10.2.1 下载仓库快照

随着你的 Git 超能力不断增长，你将会更少从 GitHub 下载压缩包。如果你希望将代码共享给想要 .zip 压缩包的人（或许甚至是你自己的项目），那么仍然可以使用这个选项。

为了下载项目的 .zip 压缩包，你需要完成以下步骤。

(1) 访问你想要下载代码的项目页面。
(2) 找到并点击 Download ZIP（下载压缩包）。这个按钮（图 10-12）通常位于本地克隆 URL 旁边，或者位于 GitHub 桌面应用（适用于 Windows 和 OS X）中。

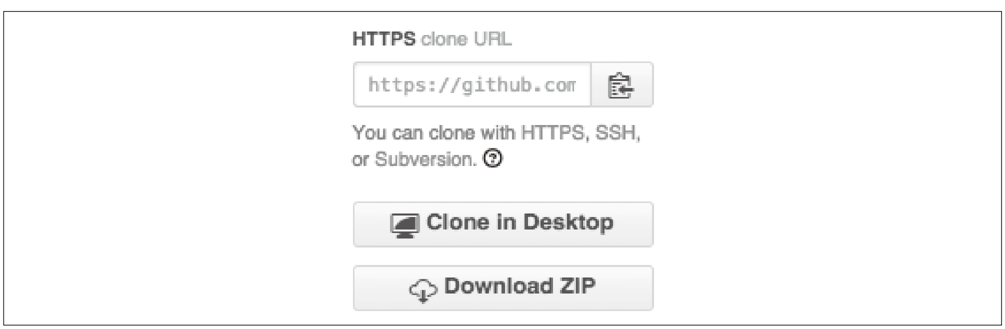


图 10-12：下载一份仓库快照

下载的文件压缩包将根据你下载的项目和分支命名。要想更改你下载的分支，请完成以下步骤。

(1) 找到并点击仓库主页左上方的分支下拉菜单（图 10-13）。

(2) 选择你想要下载的分支。等待页面刷新。
(3) 找到并点击 Download ZIP（下载压缩包）按钮。

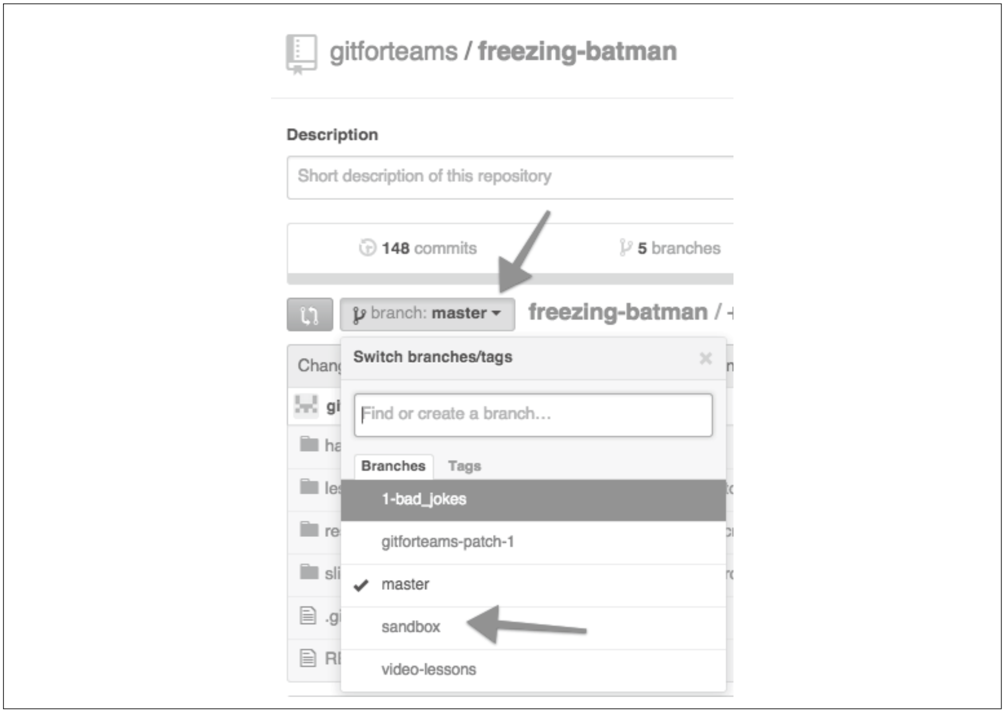
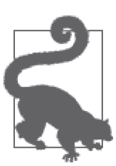


图 10-13：首先选择一个不同的分支，以改变下载的分支

用户界面上不会提示你正在下载不同的分支，但是，文件名将会反映分支的名字（repository_name-branch_name.zip）。

10.2.2 在本地工作

连接到别人的 GitHub 项目的流程与使用你自己的项目几乎完全相同，除了你没有那个项目的写入权限之外（当然，除非你被添加到那个项目团队）。在本节中，你将会学到如何创建一个本地克隆。我使用这个技巧来编写“Git 团队协作”网站（http://gitforteams.com/），此网站使用静态网站生成工具 Sculpin。



Sculpin 入门

Sculpin 是一个使用 PHP 构建的静态网站生成工具。本节中的说明尚不足以让你能够运行一个网站。如果你对尝试 Sculpin 有兴趣，请从入门指南（https://sculspin.io/getstarted/）开始学习。

在这种情况下，我希望在网站中使用 Sculpin 模板的本地副本。尽管我是 Sculpin 项目的志愿者，这个仓库只是针对我的网站的。我不可能在本地副本中为项目做出贡献。然而，我

仍然希望保持一个主项目的连接，以便轻松地将最新的更新加入我的网站。尽管这些命令特定于 Sculpin 项目，但你也可以使用你的项目来替换 URL。

第一步是创建一份项目的本地克隆（例 10-3），如下所示。

(1) 前往你想要下载的仓库的项目主页。
(2) 找到并点击“复制到剪贴板”图标（图 10-14）以获得仓库 URL。
(3) 打开一个命令窗口（或 Windows 上的 Git Bash 窗口），前往你想要将项目下载到的目录。
(4) 使用 `clone` 命令和你在第 2 步中复制的 URL 来创建一份项目仓库的本地副本。可选地，将目录名称添加到命令末尾。
(5) 将目录名称改为项目相关的名称。你也可以通过在上一步中在命令末尾添加新目录的名称来完成这一步。
(6) 前往本地仓库。

例 10-3　创建仓库的克隆

```
$ git clone https://github.com/sculpin/sculpin-blog-skeleton.git
$ mv sculpin-blog-skeleton gitforteams.com
$ cd gitforteams.com
```

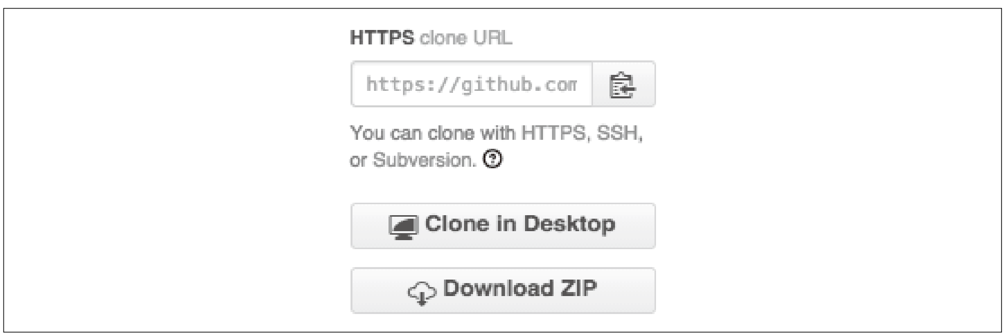


图 10-14：“复制到剪贴板”图标位于下载按钮下方

第二步（例 10-4）创建了一个上游，或者说“供应商分支”，它将独立于你项目中特有的更改。你可以通过这个分支来更新主项目的任何更改。对于我正在进行的项目，它的默认分支是 *master*。你可以选择更适合你的名称，有时我使用项目名称，有时我使用通用的别名 *upstream*。我不认为使用哪一个更好（尽管 Shakespeare 会对我的命名怪癖有意见）。通过移动分支而不是新建一个分支，我维持了本地分支和远程仓库的关系。如果你倾向于在主分支上工作，那么也可以选择重新创建 *master* 分支。

例 10-4　创建一个上游分支

```
$ git branch --move master upstream
$ git checkout -b master
```

最后一步是为项目的工作副本添加一个远程仓库（例 10-5）。这个新的远程仓库将会存放所有你在你的项目实例中做出的更改。Sculpin 项目不应该记录所有我为“Git 团队协作”网站做出的更改，但我需要跟踪这些更改。在现实中，我在 Bitbucket 上以私有仓库的方式保存“Git 团队协作”仓库。我不使用 issue 跟踪工具，而只是在仓库中丢掉这些更改，然

后在提交后上传，就像一个备份方案一样。我没有利用 Bitbucket 提供的功能，但这能让我不用过多思考。

当第一次克隆项目时，远程名称 *origin* 被分配给远程仓库。我们将要交换 *upstream* 的别名，因为约定是使用 *origin* 作为最接近我们自己的仓库的别名。

为了准备添加新的远程，你将需要确定它的 URL。如果你现在还没有设置一个远程仓库，请遵循本章之前的创建项目的步骤，并确保项目在初始化过程中没有添加任何文件。一旦你创建完新项目，请遵循屏幕上的说明来添加仓库中的远程信息，然后上传更改。例如，如果你的 GitHub 用户名是 gitforteams，而你的新仓库的名称是 superhero-freda，那么你需要按照例 10-5 中那样添加远程仓库。

例 10-5　为工作副本添加一个远程仓库

```
$ git remote rename origin upstream
$ git remote add origin https://github.com/gitforteams/superhero-freda.git
$ git push -u origin master
```

现在，你同时拥有一个名为 `upstream` 的分支和一个名为 *upstream* 的远程仓库。

定期检查上游仓库以获得更新（例 10-6）。签出你为项目分配的上游分支并拉取更改以完成这一步。

例 10-6　检查上游分支以获得更新

```
$ git checkout upstream
$ git pull --rebase=preserve
```

假设主项目有更新，你可以阅读这些更改来决定是否想要将这些更改并入你自己的项目（例 10-7）。

例 10-7　比较上游的更改和你本地的工作

```
$ git diff master upstream
```

或者你可以在 `log` 命令后添加一些华丽的参数，查看特定提交的摘要，如下所示。

```
$ git log --cherry-mark --left-right --oneline master...upstream
```

我们以前看到过这个命令的变种，真正新鲜的是 `--cherry-mark --left-right`。这些参数在提交开头添加一个符号，表明这个更改是列表中第一个分支（向左）还是第二个分支（向右）引入的。

一旦你理解了这些更改，你就可以更新你的分支，与上游更改保持一致（例 10-8）。如果更改已经存在，而你今天的工作尚未开始。换句话说，你应该通过将上游仓库上的更改变基到你自己的分支上，来更新你的工作分支。（像我之前提到的，如果你单独工作，那么同样可以使用合并，如果你觉得这比 `rebase` 更容易使用，我不做评判。）

例 10-8　并入上游更改

```
$ git checkout master
$ git rebase upstream
```

如果冲突发生，每次解决一个。第 6 章还介绍了其他处理变基冲突的建议。

10.3　为项目做出贡献

你已经决定更进一步，向项目提交一个贡献。很好！恭喜你！这和你之前所做的并没有很大的不同。主要的区别在于你将提交一个拉取请求，在合并到主项目前要经由其他人评审。

10.3.1　使用issue跟踪更改

在公开项目中，issue 通常被找到 bug 的用户打开。很少的一些贡献者会创建 issue 来记录他们想要贡献的新功能，或者他们在开发时感兴趣的设计更改。



派生项目的 issue 是默认关闭的

对于仓库间的派生，issue 是默认关闭的。如果你想要跟踪你的派生项目的 issue，可以在设置页中启用这项功能。

要创建一个 issue，请完成以下步骤。

(1) 前往项目页面。

(2) 找到并点击标签为 Issues 的选项卡。该选项卡位于顶部的导航栏中（图 10-15）。系统会将你重定向到 issue 页面。

(3) 找到并点击 New issue（新建 issue）按钮。该按钮位于屏幕的右侧（图 10-16）。系统会将你重定向到一个 issue 创建表单。

(4) 输入标题、想要解决的问题的描述（图 10-17）和提交这个拉取请求要解决的 issue 的工单编号。描述问题时越具体，它越有可能得到解决。

(5) 当你对你的 issue 描述满意时，找到并点击 Submit new issue（提交新 issue）按钮。

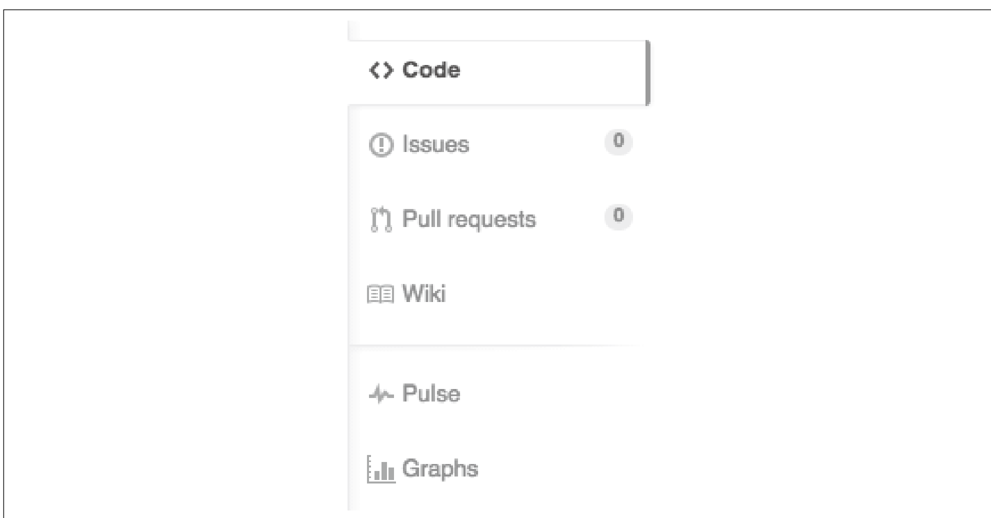


图 10-15：Issues 的导航图标

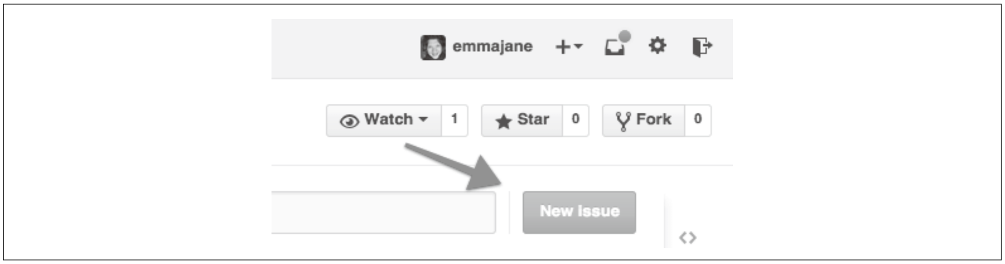


图 10-16：新建 issue 的导航按钮

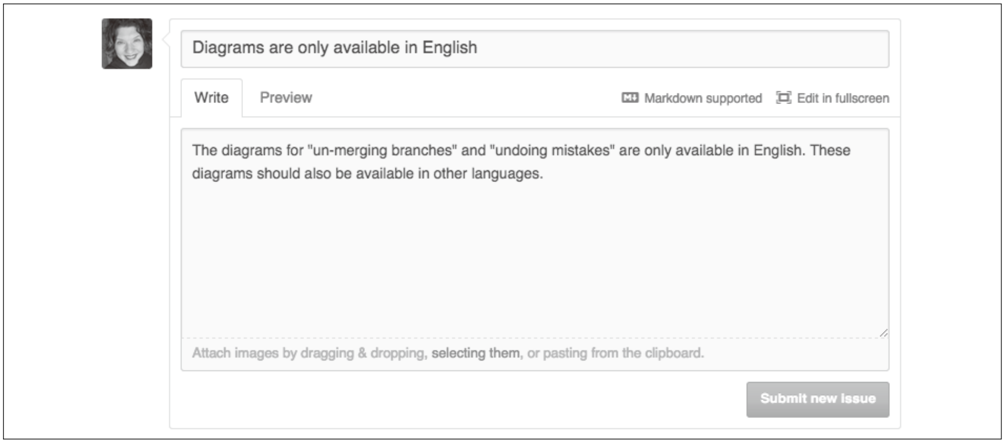


图 10-17：新建一个 issue

在 issue 创建后，你现在可以继续创建解决这个 issue 的拉取请求。

10.3.2 派生项目

如果你想要将你的更改贡献回去，需要完成以下步骤。

(1) 前往项目页面。

(2) 找到并点击 Fork（派生）按钮。仓库将会被派生，而你将会在你自己的账户下看到一个仓库的副本。

现在，你可以将项目的这个副本克隆到你的本地计算机，就像你在 10.1.3 节中所做的那样。一旦仓库下载后，你可以在项目中进行更改，将它们提交到你的仓库，然后将它们推送回你的远程仓库的派生副本。

一旦你想要合并到主项目的这些更改被推送回 GitHub 之后，你就需要创建一个拉取请求。

10.3.3 创建拉取请求

当你创建项目派生时，GitHub 维护了一个上游项目的连接。这允许你轻易地将你派生仓库中的更改发送到主项目。

要发起拉取请求，请完成以下步骤。

(1) 前往你派生的仓库的项目页面。
(2) 找到并点击拉取请求按钮（图 10-18）。该按钮位于项目描述的左上方，标题的下方。系统会将你重定向到可用于拉取请求的分支摘要。如果没有显示四个下拉菜单，在继续前点击 compare across forks（在派生间比较）。
(3) 在分支列表中，从最后的下拉菜单中选择你想要提交到上游项目的选项（图 10-19）。你的分支和上游分支间的差异将会显示。
(4) 找到并点击 Create pull request（创建拉取请求）按钮（图 10-20）。一个新的表单将会打开。
(5) 输入标题和一段你为什么向项目提交这个更改的描述（图 10-21）。
(6) 找到并点击 Create pull request 按钮来完成让你的更改进入上游项目的请求。



图 10-18：拉取请求按钮位于项目标题下方

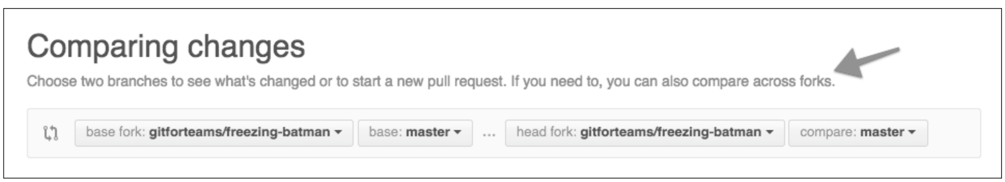


图 10-19：在拉取请求中选择你想要提交到上游项目的分支

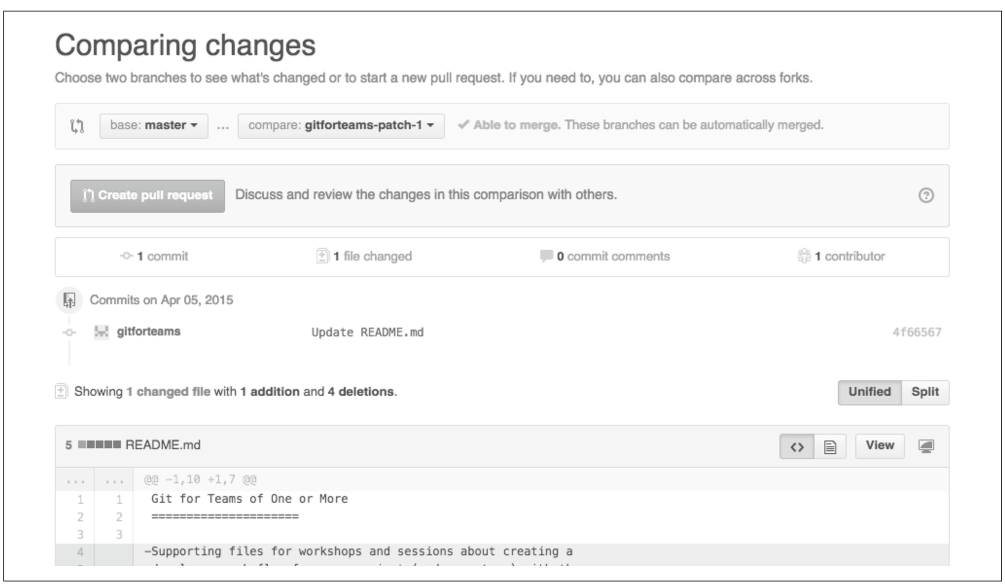


图 10-20：为了启动拉取请求过程，找到并点击 Create pull request 按钮

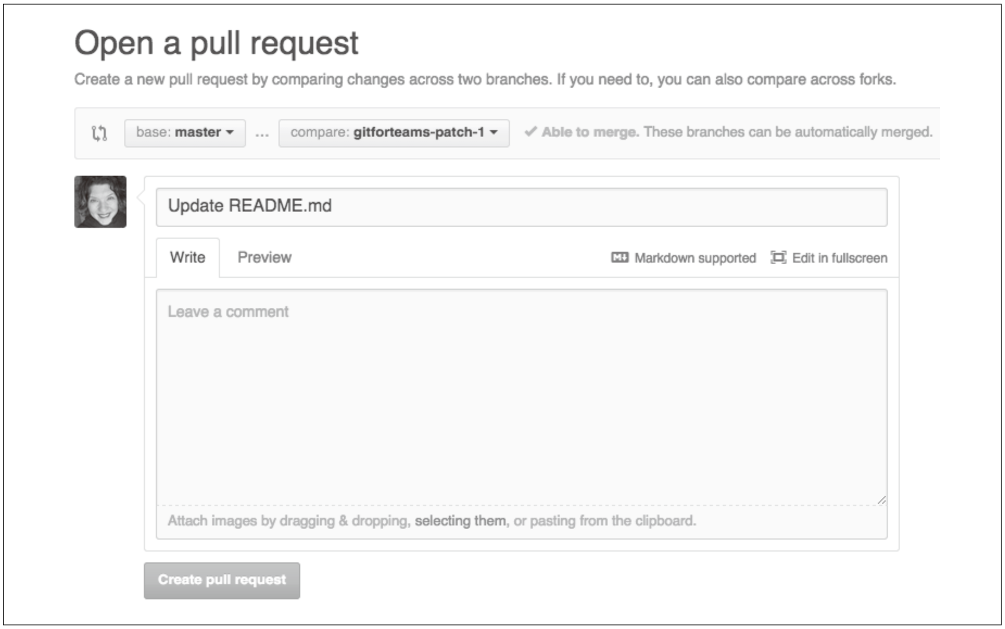


图 10-21：输入标题和摘要来解释提议更改的原因

一旦你完成了拉取请求之后，项目的维护者将会在他们的 GitHub 项目界面上收到通知和邮件提醒（如果已启用通知）。

10.4　运营你自己的项目

在 GitHub 上运营一个项目的技术部分非常容易。GitHub 为你提供了 issue 队列、附带的文档页面（wiki）、通过拉取请求支持外来代码更改以及为仓库授予写入权限的能力。因此，困难在于社交部分，如何创建一个以你的软件项目为中心的消费者和贡献者社区。你应该回到第 2 章来回想起如何运营好一个项目。

10.4.1　创建项目仓库

我的大多数公开 GitHub 仓库都很小，比如各种会议的演示文稿。我并不期望会有定期的贡献者，尽管如果人们有兴趣提交一个新的修复，我会很高兴接受贡献。如果你正在制作一个软件包，其他人更有可能有兴趣为项目做出贡献。如果你正在编写一个你认为会帮到更多人的库或软件包，则不应该将它放在你的私人账户下，而应该使用组织。不使用私人账户可以让开发者对这个项目产生更大的归属感，并专注地做出贡献。

要创建新项目，请完成以下步骤。

(1) 在顶部菜单中，点击 + 符号。
(2) 点击 New repository（新建仓库）。系统会将你重定向到新建项目表单。
(3) 在 Owner（所有者）标签下，点击你的账户并将它更改为你的组织。

(4) 输入仓库名称。一般来说这个名称应该与项目相同，如果组织中只有一个仓库。
(5) 为你的项目输入一段精简的描述。
(6) 点击 Create repository（新建仓库）。

你的新仓库已经创建，现在你可以开始使用这个仓库，就好像这是你的一个私人 GitHub 仓库一样。

如果项目已经存在于你的私人账户下，你可以使用以下步骤重新分配这个项目。

(1) 前往你的个人账户下的项目页面。
(2) 找到并点击 Settings（设置）链接。
(3) 找到并点击 Transfer（转移）按钮。此时将出现一个模式窗口。
(4) 输入仓库名称，新的所有者的组织名或账户名。
(5) 点击 I understand, transfer this repo（我理解将会产生的后果并转移仓库）。

你的项目将会被重新分配新的账户。

根据你的治理规则，你需要决定是准备使用拉取请求还是继续将工作直接提交到项目。两种方式都有各自的优点，但它们也遵循不同的领导力模型（直接提交更快；但如果你还要提交需要评审的拉取请求，对所有贡献者来说更加公平）。

10.4.2　授权共同维护

为了分担维护的重担，你可以将仓库的写入权限授权给他人。这是一个重大的责任。你应该提前决定你想要怎样处理棘手的问题，比如代码方向上的分歧以及其他类型的不良行为，比如对其他贡献者不友好。假设你已经想明白了这些复杂的决定，则可以通过以下步骤添加项目的贡献者。

(1) 前往项目页面。
(2) 在页面右上方的工具链接中点击 +，然后选择 New collaborator（新建协作者，图 10-22）。
(3) 系统将提示你输入密码，然后点击 Continue（下一步）。
(4) 输入你想要分配的共同维护者的 GitHub 用户名（图 10-23）。

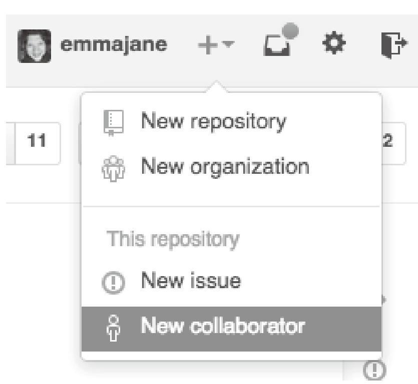


图 10-22：前往项目的协作者页面

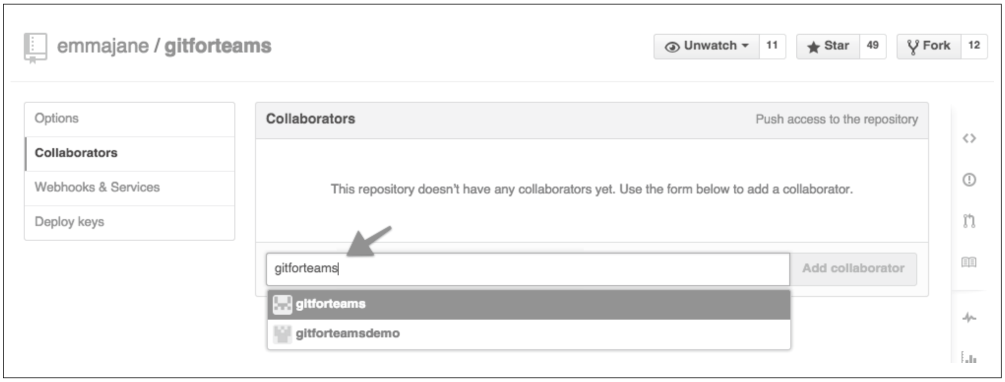


图 10-23：在项目中添加协作者

你指定成为共同维护者的成员现在拥有与你相同的管理权限。你或许希望指定一个维护备忘单，确保你们的决定对于所有社区成员来说是一致的。

要想移除一个协作者，请遵循之前概述的说明。在协作者姓名右边，点击 x 符号（图 10-24）。这个协作者将不再拥有仓库的提交权限。



图 10-24：从项目中移除协作者

10.4.3　评审并接受拉取请求

恭喜你！你已经收到了项目的第一个拉取请求。GitHub 为你提供了一个易于使用的界面来评审收到的拉取请求。你可以在这里评审这个请求，立即拒绝或接受这个拉取请求。

如果接受拉取请求会导致合并冲突，那么 GitHub 会通知你，并且在这种情况下会禁用接受传入请求的按钮。



试一试给你自己提交一个拉取请求

你也可以通过派生你自己的工作，然后给自己提交一个拉取请求来测试这个功能。

10.4.4 发生合并冲突的拉取请求

如果拉取请求一定会导致合并冲突，你将无法在 Web 界面上接受这个拉取请求。相反，你需要下载这个分支，在本地解决冲突，然后将新的分支推送到项目仓库。

第一步是签出你想要接受这个拉取请求的分支。例如，你或许希望将它并入你的项目主分支，如下所示。

```
$ git checkout master
```

目前你的分支对贡献者的仓库毫无所知。在下载提议的更改前，你需要将它添加为远程仓库。与其使用我们过去的通用别名（如 *origin* 或 *upstream*），不如乐观些，使用贡献者的 GitHub 用户名。这将确保你准备好以后会接受他们更多的更改。

在下面的例子中，使用拉取请求的分支中合适的值替换 <*username*> 和 <*repository-name*>。

```
$ git remote add username git://github.com/<username>/<repository_name>
```

在添加远程仓库后，你现在必须下载贡献者的工作，如下所示。

```
$ git fetch username
```

这个分支现在将被下载供本地评审使用。你应该使用第 7 章中实行同行评审的指南。如果代码存在问题，你可能需要向评审者提供反馈并要求他提交新的拉取请求。参考你的治理模型来决定你自己来更新是否合适，或者你是否需要重新打开该 issue 以进行进一步的研究。. 经验法则是：如果贡献者会从这个工作中学到什么，那么给他们这个学习的机会。如果这只是一个愚蠢的小错误（拼写错误或者违反了代码规范），或许你自己修改会更合理一些（仍然鼓励原作者），而不是因为一个微小的修复而拒绝这个拉取请求。只要有可能，减少代码需要的往返，并尊重贡献者的意图。

当你对提议的更改感到满意时，可以将它合并到项目的主分支，如下所示。

```
$ git merge --no-ff username/branch_name
```

然而，如果你希望进行一些修改来清理几个空格问题，或者修复一个拼写错误，你可以选择添加 `--no-commit` 参数。如果你决定每个更改必须经历拉取请求环节，那么在你的项目中使用这个选项或许会不合适，如下所示。

```
$ git merge --no-ff --no-commit username/branch_name
```

不管你使用什么方法，一旦分支合并后，就可以将更新后的 `master` 分支推送到服务器上，如下所示。

```
$ git push origin master
```

更改将会出现在项目的主仓库中。

如果你发现你的项目经常使用拉取请求，频繁地需要处理合并冲突，那么 Hub（https://hub.github.com/）能够帮到你。这是一个命令行外层应用，允许你享受到在命令行中执行更多任务的便利，而无需在 GitHub 的 Web 界面和 Git 之间切换。

10.5 小结

在本章中，你学会了如何在单人团队中作为其他项目的消费者、贡献者以及项目负责人来使用 GitHub。

- 作为仓库的所有者，你可以选择直接在上面做出贡献。
- 作为项目的负责人，你可以选择直接在项目中提交，或者通过个人仓库来传递你的贡献以示公平。
- 项目中的 issue 可以用于跟踪新功能或者 bug。issue 以对话的形式出现，或许会因此产生一个拉取请求。
- 拉取请求是合并外部仓库中的分支或非主分支的请求。它可以由拥有仓库写入权限的任何人完成。
- 如果拉取请求不会导致合并冲突，它可以通过网页界面来完成，否则，你将需要下载相关的分支，在本地合并请求并将结果推送回主项目仓库。

尽管本章主要讲述公开仓库，你同样可以将本章中学到的技巧用于私有仓库。

有关使用 GitHub 的更多信息，你可以阅读 Peter Bell 和 Brent Beer 合著的 *Introducing GitHub*（http://shop.oreilly.com/product/0636920033059.do）。

第11章

Bitbucket上的私有团队工作

Bitbucket 是一个流行的代码托管系统，由 JIRA 的开发者创建。它有接近三百万用户，用户数量比 GitHub 少一些，但对于小团队来说，它有两个很大的优点：免费的私有仓库和分支特有的权限控制。除了这些功能之外，我认为 Bitbucket 的界面非常简单直接，而文档却很复杂。它所承诺的易用性需要一段时间才能使内部团队运转顺利。

完成本章学习之后，你将能够在 Bitbucket 上完成以下事项。

- 以个人开发者的身份起步
- 与其他开发者共享你的仓库
- 限制给定项目分支特有的访问控制

本章不是全面的 Bitbucket 指南，而是你或许想在团队中使用的一些重要功能的概述。

喜欢通过视频教程学习的读者可以参考本书附带的系列视频，Collaborating with Git（http://shop.oreilly.com/product/0636920034872.do）。

11.1 非公开项目的项目治理

Bitbucket 仓库与 GitHub 相比使用不同的默认选项，这带来了有趣的影响。根据你的视角，你或许会认为它们是“谨慎的”或“反社交的”。默认情况下，Bitbucket 假设你即将创建的仓库是私有的，而这个仓库的派生同样也是私有的。这与 GitHub 的选择（公开仓库和公开派生）完全相反。当 GitHub 使用了“社交化编程”这个词的时候，Bitbucket 采用了一个非常不同的做法，但它不只是社交的对立面那么简单。也就是说，Bitbucket 并不是反社交的。相反，它只是默认采取了谨慎的做法。

对于私有仓库和公开仓库来说，尽管你用来将代码从一处转移到另一处的命令是相似的，

但是当人们收到邀请参与项目时，给他们的政治感觉是不同的。开源项目倾向于遵守仓库整体的访问控制。只有非常少的维护者可以更新代码的任何部分。当然，代码如何被项目接受的约定各有不同，但是一般来说，一个提交出现后，经过某个评审期，然后代码被项目的主仓库接受。而私有仓库倾向于拥有非常细致的治理要求。有时这些要求是监管机构列出的，例如进行金融交易的公司必须符合支付卡产业（PCI，Payment Card Industry）规定，制造生物医疗设备的公司必须遵从某些规定。在某些情况下，这些规定对审计和接受代码库中的贡献有着严格的要求。

目前，与 GitHub 相比，Bitbucket 提供了更多更细粒度的访问控制。在 Bitbucket 上，你可以阻止个人、一群人推送到特定的分支和整个仓库。如果你喜欢使用 Subversion 中分支特有的访问控制，你的团队将会发现这个功能非常有用。GitLab 同样提供了其中的一些功能，将在第 12 章中介绍。

11.2 开始使用

在本节中，你将会学到如何在 Bitbucket 上创建一个账户和你自己的私有仓库。团队中所有开发者应该在与你协作项目前完成本节中所述的步骤。

11.2.1 创建账户

Bitbucket 的注册过程十分直截了当，步骤如下。

(1) 前往 https://bitbucket.org。
(2) 找到并点击标签为 Get started（开始使用）的按钮（图 11-1）。（可能有多个按钮。任意一个皆可。）

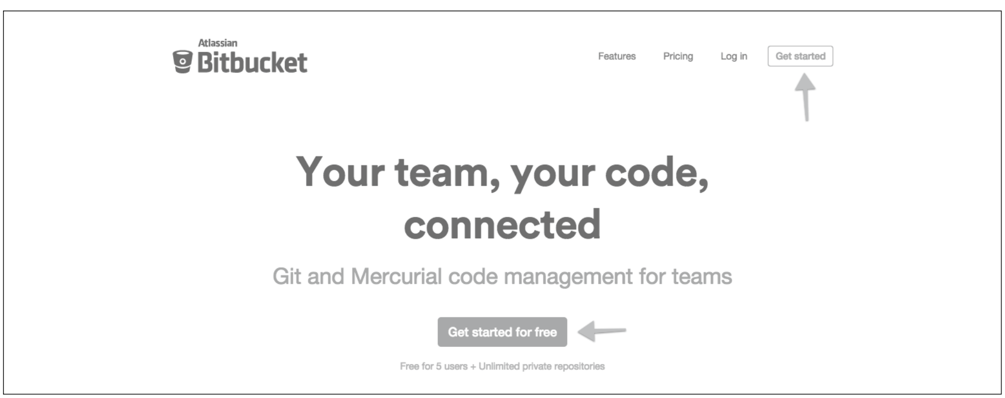


图 11-1：在主页上找到并点击其中一个 Get started 按钮

你可以选择创建新账户或使用你的 Google 账户注册，步骤如下。

(1) 输入你的姓和名。这两个字段是可选的。
(2) 输入你喜欢的用户名。Bitbucket 会告诉你这个名称是否已被占用。
(3) 输入一个安全的密码。

(4) 输入一个合法的电子邮件地址。
(5) 选择一个方案。默认情况下，免费的个人账户方案会选中，它适用于个人和非常小的团队。
(6) 选中复选框，确认你不是机器人。如果 Bitbucket 认为你不是人类，那么你可能需要输入 CAPTCHA 验证码。
(7) 选中隐私策略和客户协议的复选框。显然，你应该点击这些链接并阅读你即将签署的协议。
(8) 当你完成了所有字段后，点击 Sign up（注册）以继续（图 11-2）。
(9) 你将会收到一封邮件来确认你的电子邮件地址。点击 Confirm this email address（确认此电子邮件地址）。

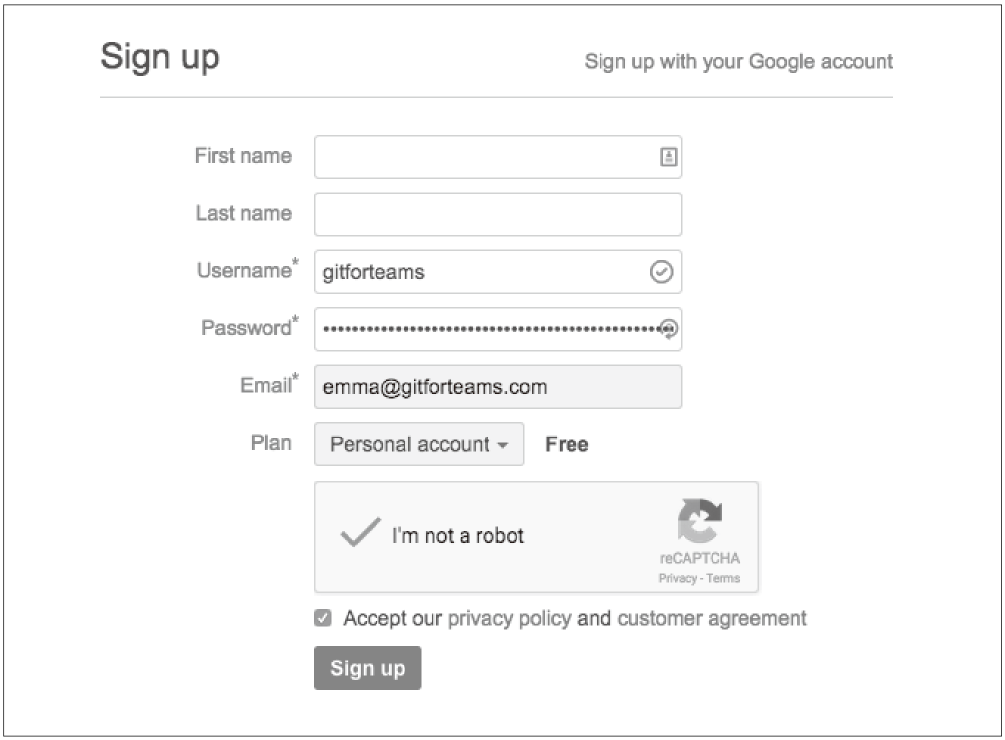


图 11-2：完成注册表单中的每一个字段并点击 Sign up

现在，你的账户已经创建并可以使用。但是，为了节省以后的时间，你还应该添加你的 SSH 密钥，这样以后每次使用私有仓库时不需要每次都验证一次。

完成以下步骤，将你的 SSH 密钥添加到你的账户中。

(1) 使用附录 D 中的说明，找到并复制你的 SSH 公钥。
(2) 前往你的 Bitbucket 账户信息中心。
(3) 在 Bitbucket 网站的右上角，找到并点击用户图标。
(4) 在下拉菜单列表中，点击 Manage account（管理账户）。

(5) 在侧边导航栏中，找到并点击 SSH keys（SSH 密钥）。
(6) 点击 Add key（添加密钥）。此时将会显示一个模式窗口。
(7) 在表单 Key（密钥）字段中，粘贴你的 SSH 公钥。
(8) 点击 Add key。

你的 SSH 密钥已经添加到你的 BItbucket 账户中。

11.2.2　在欢迎页面创建私有项目

在创建账户后，Bitbucket 将会立即带你来到欢迎页面（图 11-3）。这个页面一直可以通过 https://bitbucket.org/welcome 访问。

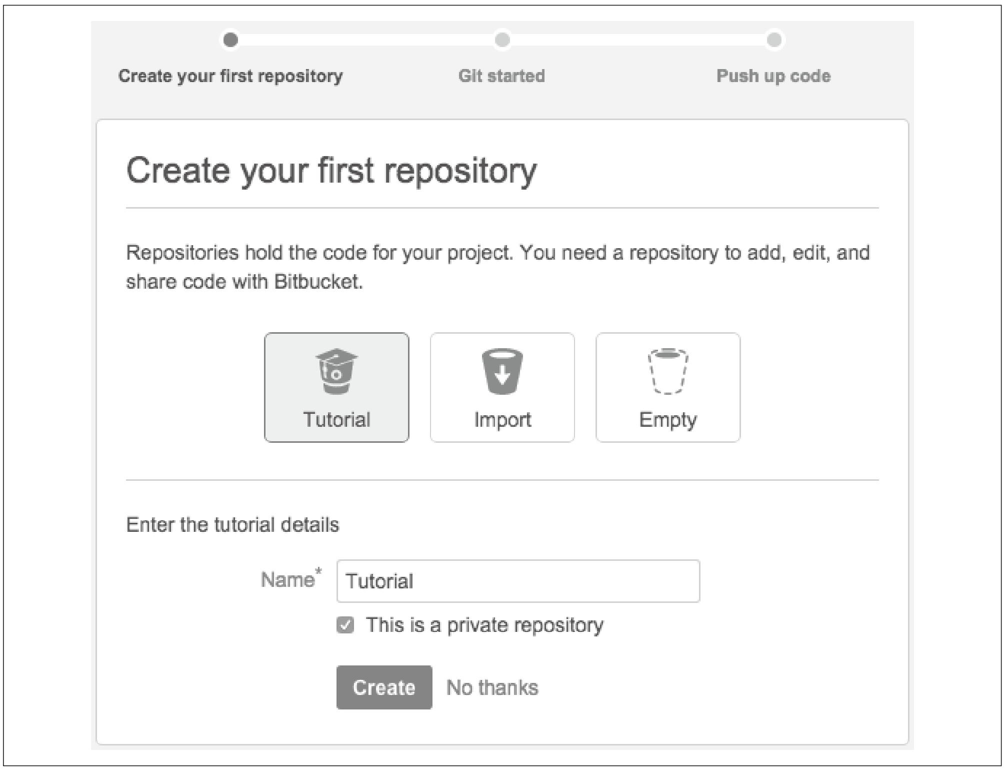


图 11-3：在完成注册表单后，将会显示开始使用的欢迎页面

要创建新仓库，请完成以下步骤。

(1) 点击标注 Empty（空仓库）的虚线桶状图标。
(2) 输入仓库的名称，如 *johannes*。
(3) 保持 This is a private repository（这是一个私有仓库）复选框选中。
(4) 点击 Create（新建）。你的新仓库已经创建。
(5) 点击 Done（完成）。此时将会显示仓库的设置页面。

一旦你完成了这些步骤，请继续阅读 11.2.4 节。

11.2.3 从信息中心创建私有项目

在登录 Bitbucket 账户后，将会显示一个项目摘要信息中心（图 11-4）。在这个信息中心中你可以概览你的每个项目的动向，以及创建一个新仓库。

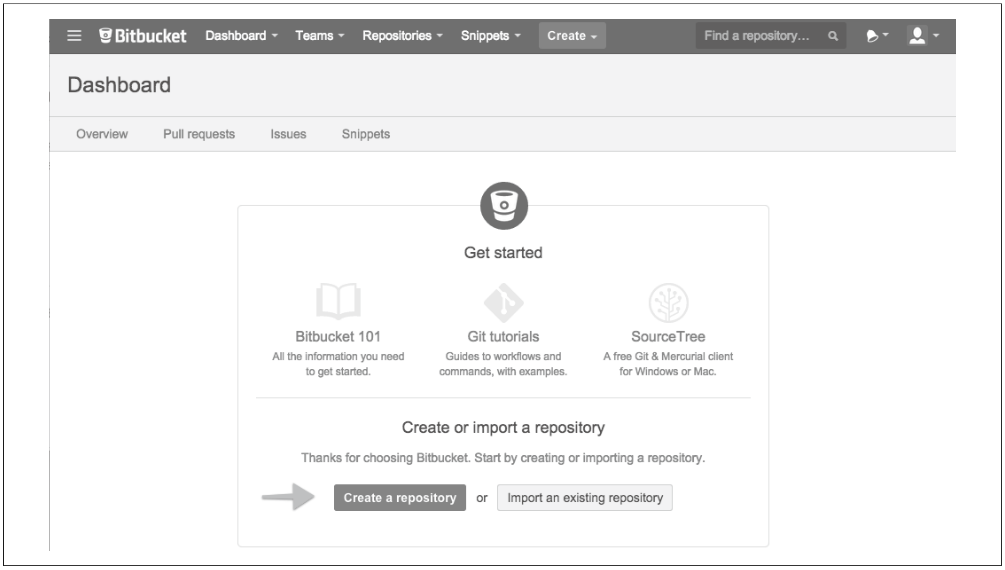


图 11-4：信息中心同样给出了如何创建一个新仓库的说明

如果你是从信息中心（这也是当你完成验证后的首页）起步的，请完成以下指示来创建新仓库。

(1) 找到并点击 Create a repository（创建仓库）按钮。此时将会显示图 11-5 中显示的表单。
(2) 输入仓库名称，如 *junio*。
(3) 可选地，输入仓库的描述。
(4) 保持下列默认设置不变。

- 访问层级（如果这是一个私有仓库，复选框应该被选中）
- 派生［下拉菜单应该被设置为 Allow only private forks（只允许私有派生）］
- 仓库类型（单选框应该被设为 Git）

(5) 你可以选择打开 Issue tracking（issue 跟踪）或 Wiki 页面。我很少在个人项目中启用这些选项，因为你只是想使用 Bitbucket 作为一个代码的远端备份，而不是一个项目管理工具。
(6) 最后，找到并点击 Create repository（创建仓库）。

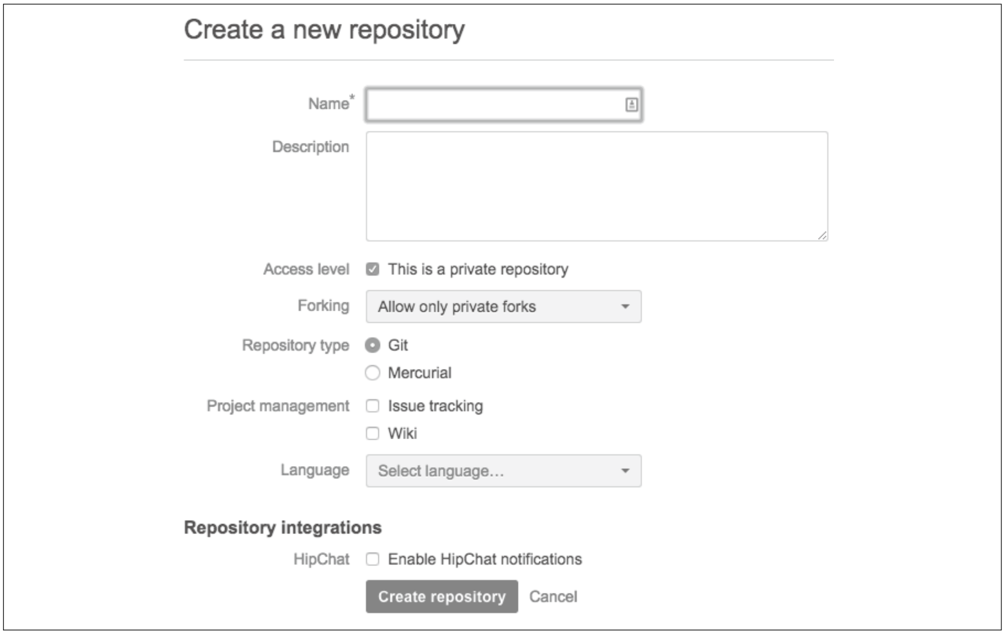


图 11-5：创建新仓库的表单同样有一些共享相关的设置项

你的新仓库已经创建，此时将会显示仓库的配置页面，请继续阅读 11.2.4 节。

11.2.4　设置你的新仓库

图 11-6 显示的是设置页面。

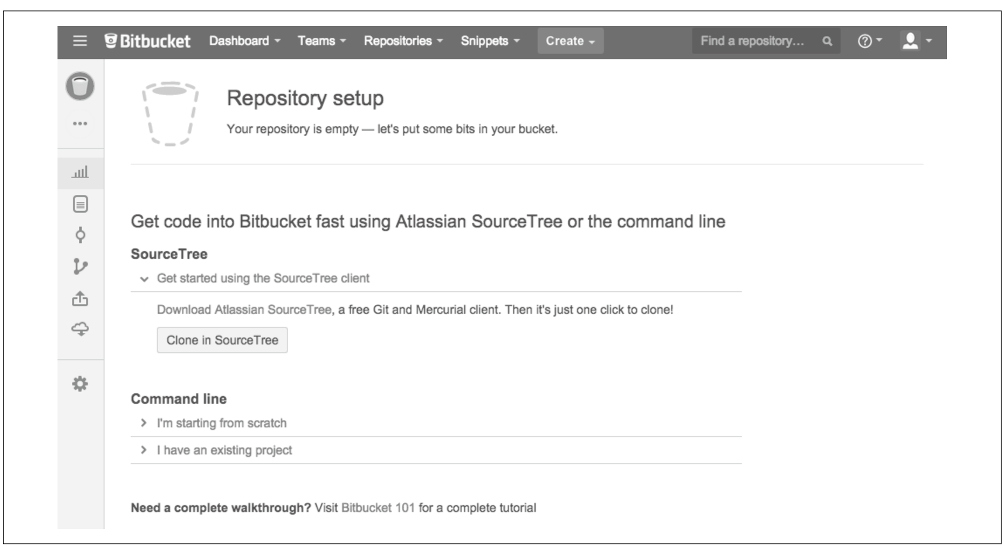


图 11-6：提供了图形界面和命令行式的设置指令（新项目或现有项目）

假设你已经完成了本书之前的练习，那么现在应该已经有了一个本地仓库，或者知道如何创建一个！我发现在 Bitbucket 上设置新项目时，最后一组说明（图 11-7）最为有用。

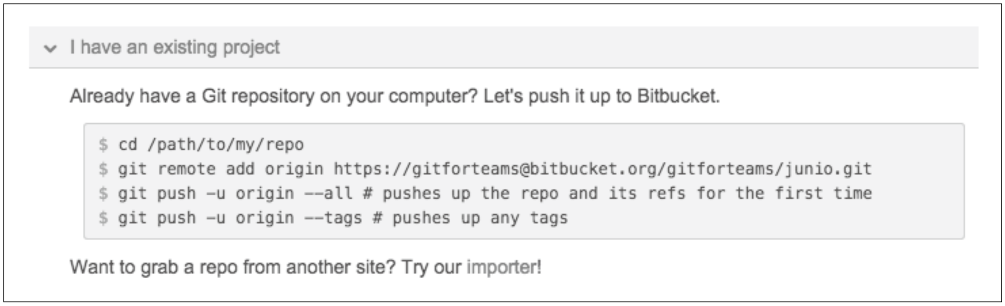


图 11-7：将现有项目连接到 Bitbucket 的设置指令

要连接你的本地仓库和 Bitbucket 上的新仓库，请完成以下步骤。

(1) 找到并点击 I have an existing project（我已有项目）链接。一组附加的说明将会出现在屏幕上。
(2) 在命令行中，前往一个本地 Git 仓库。如果它已经连接到一个不同的托管系统也没关系，你可以拥有多个远程仓库连接。
(3) 复制并粘贴说明中 `git` 开头的这些命令（例 11-1）。

例 11-1　Bitbucket 上的样例说明，添加新创建的仓库为本地仓库的远程

如果该仓库已经连接到一个远程仓库，你或许需要使用 `bitbucket` 替代 `origin`。

```
git remote add origin https://gitforteams@bitbucket.org/gitforteams/junio.git
git push -u origin --all # pushes up the repo and its refs for the first time
git push -u origin --tags # pushes up any tags
```

使用你的指令，而不是我的

不要复制上面的代码片段中的指令。复制 Bitbucket 在你刚创建的仓库的摘要页面上提供的指令。

现在，你已经准备好作为个人开发者在私有仓库中工作。你可以将你的代码更改随时推送到 Bitbucket 上。并且因为它是一个私有仓库，你永远不需要担心破坏公共历史！如果你变基了一个分支，而 Bitbucket 拒绝接受这个分支的新版本，请在你即将输入的命令后加上 `--force` 参数，如下所示。

```
$ git push --force
```

如果你是在一个团队中工作，那么更优雅的写法应该是下面这样的。

```
$ git push --force-with-lease
```

我们在下一节中将会探索 Web 界面。同时，你或许会从看到的选项中发现一些价值。如果你在前几节中已经使用过 GitHub 或 GitLab，我相信你会发现大量的选项都十分相似。

11.2.5 探索你的项目

一旦你的项目被推送到 Bitbucket 后，项目页面将会自动从一组指令变成一个项目浏览器。

如果你的仓库包含 README 文件，这个文件将会显示在项目主页上。图 11-8 显示了我的“Git 团队协作”网站（http://gitforteams.com/）的项目主页。

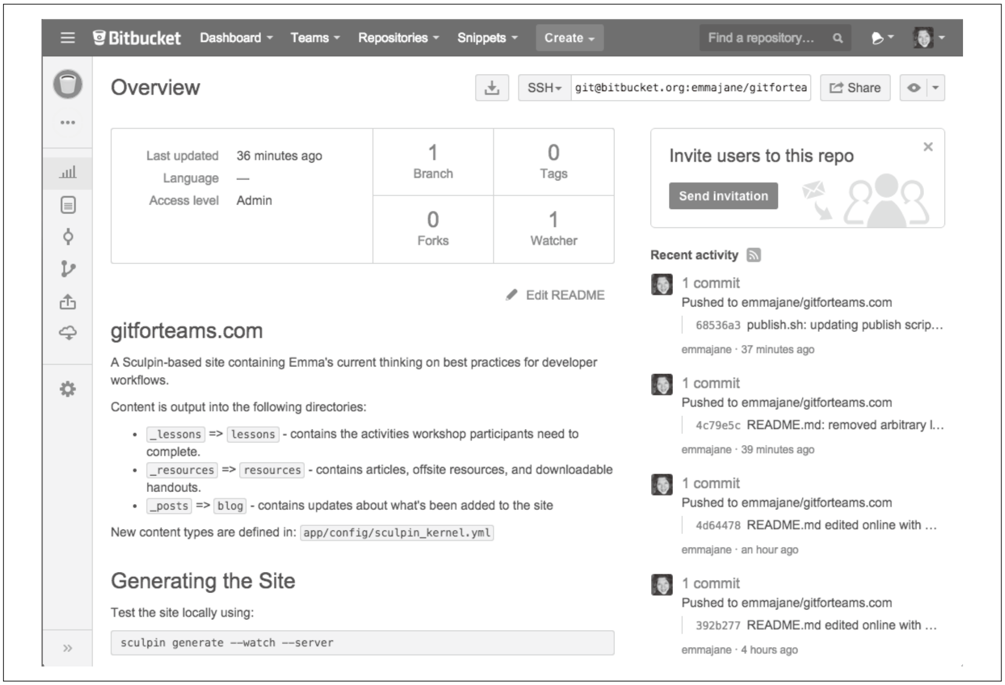


图 11-8：项目主页显示了你的站点状态摘要以及 README 文件的内容

在项目主页上，你可以找到以下摘要信息。

- 最近更新日期
- 语言（如果设置过）
- 访问层级 [将被设为 Admin（管理员），如果这个仓库是你的]
- 分支 [点击 Branch（分支）上方的数字来查看所有分支的列表]
- 标签 [点击 Tags（标签）上方的数字来查看所有标签的列表]
- 派生 [点击 Forks（派生）上方的数字来查看所有公开派生的列表]
- 关注者 [点击 Watcher（关注者）上方的数字来查看关注仓库的账户的摘要]
- 最近活动（位于右侧边栏，包括最近提交以及合并的分支）

左侧边栏有以下图标（自顶向下）。

- 项目主页链接
- 快速操作（包括克隆、创建分支和创建拉取请求）
- 概述（与项目主页内容相同）

- 源代码（仓库中所有文件的列表）
- 提交（仓库中所有的日志历史记录）
- 分支（只有在项目中推送超过一个分支后才会出现）
- 拉取请求（与个人项目无关）
- 下载（提供当前分支的压缩包列表。你也可以在此添加项目未跟踪的二进制文件）
- 设置（包括访问详细信息、仓库名称和插件）

在页面底部，有一个按钮用于展开这些图标，以显示每个图标的文本标签。一旦你展开了侧边栏，就可以通过再次点击这个双箭头来折叠它（图 11-9）。

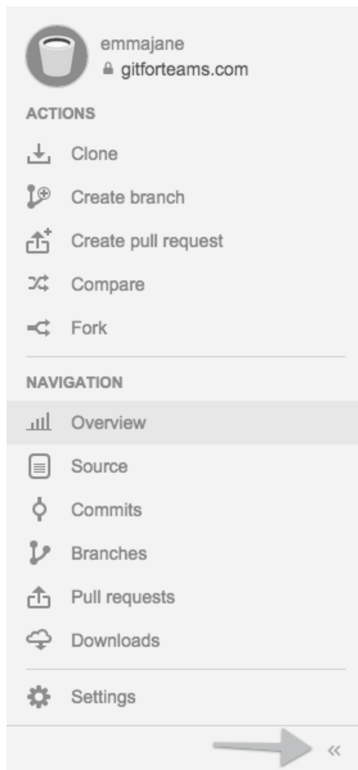


图 11-9：展开的项目侧边栏

11.2.6　编辑仓库中的文件

Bitbucket 允许你在基于 Web 的文本编辑器中编辑纯文本文件，步骤如下所示。

(1) 点击侧边栏中的 Source（源代码）链接。
(2) 前往你希望编辑的页面。
(3) 找到并点击 Edit（编辑）按钮。文本编辑器将会出现（图 11-10 或图 11-11 中项目 README 文件的编辑器）。

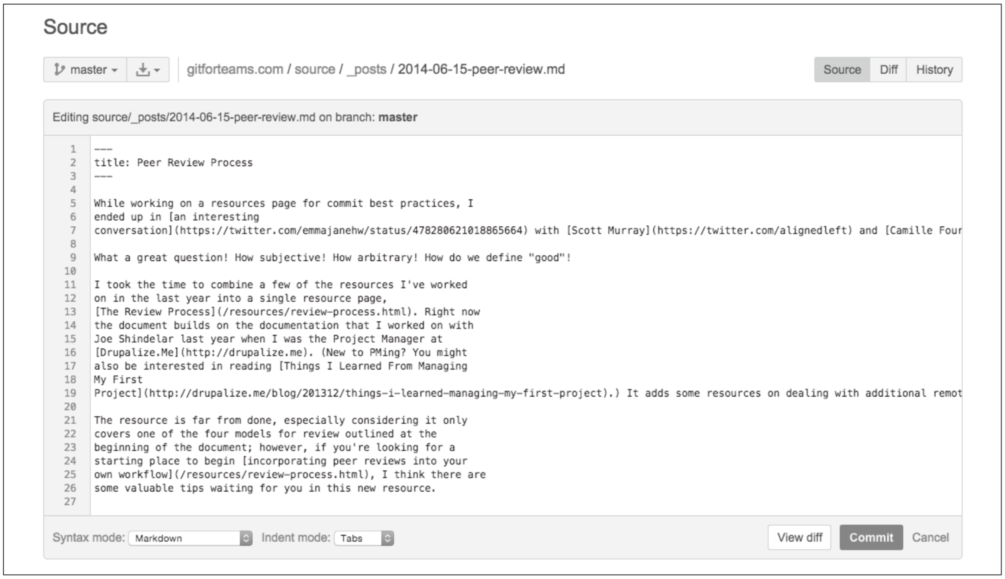


图 11-10：仓库内的文本编辑器

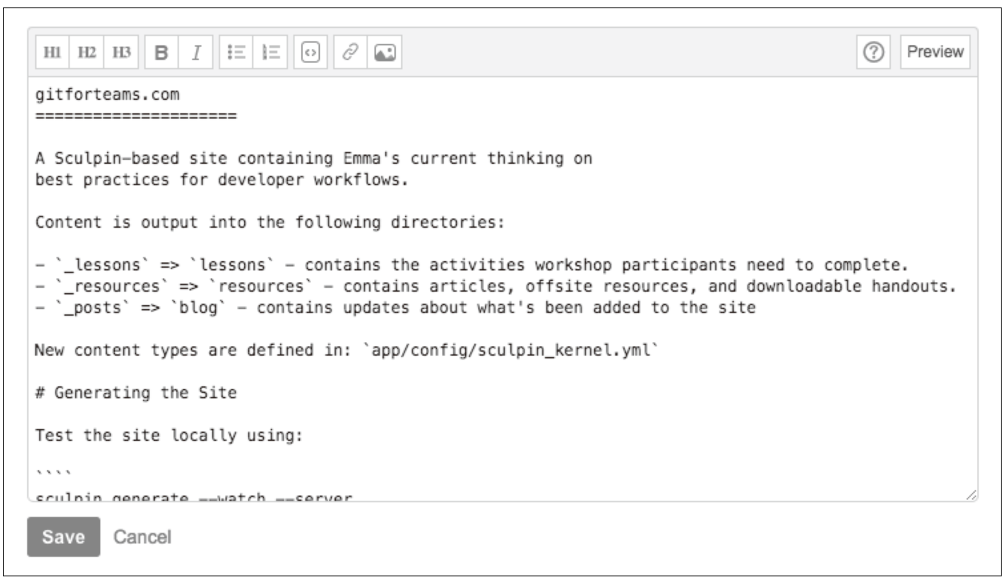


图 11-11：项目主页编辑器

(4) 在编辑器底部，确认 Syntax mode（代码高亮模式）、Indent mode（缩进模式）和 Number of spaces（空格数量，并非所有文件类型都支持）设置正确。
(5) 编辑文件以完成必要的更改。
(6) 找到并点击 View diff（查看差异）按钮。
(7) 确认更改完成且正确，没有引入多余的空格。

(8) 找到并点击 Commit（提交）按钮。此时将会显示模式窗口（图 11-12）。

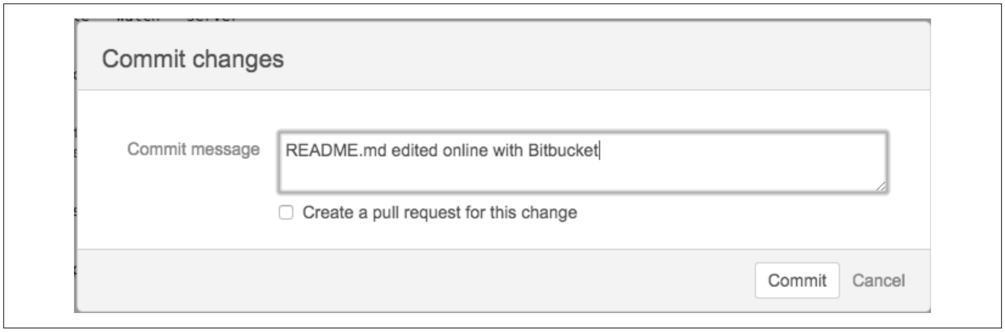


图 11-12：添加说明描述你对项目主页做出的更改

(9) 输入提交说明。你将需要添加你自己的格式。首行应该是一个不超过 80 个字符的精简描述。接下来的行可用于补充细节。
(10) 找到并点击 Commit（提交）按钮。

你的更改已经保存到 Bitbucket 上的仓库中。

将你的更改保存到 Bitbucket 后，你的本地仓库将会过时。你需要更新你的本地仓库。由于仓库完全是你自己的，你可以将更改拉取到你的本地副本而不经审查（例 11-2）。假设你遵循了本节中列出的指令，工作应该在项目的主分支上完成，最有可能是在 *master* 上。

例 11-2　将 Bitbucket 中的更改拉取到你的本地仓库

```
$ git checkout master
$ git pull --rebase
```

这些更改应该被干净地应用。但是，如果你遇到了冲突，请参考第 6 章。

你的本地仓库已是最新。

11.3　项目设置

你已经阅读本书有些时间了。甚至可能你早就开始了。因此，你知道我喜欢写关于 Git 的东西。我也知道有很多人厌恶写文档，而且觉得维护起来都是痛，所以我知道你一定非常不情愿。准备好了吗？我认为文档是保持团队关系快乐、健康的最重要的事情之一。现在，作好准备，我们继续。

记录你的过程，实现以下目的。

- 让人们更容易参与到你的团队中。
- 设置工作应该如何完成的标准。
- 作为“为什么特定的方法和命令是首选”这一讨论的起点。

就像为保龄球馆建立护栏那样，优秀的文档为你的项目建立了屏障。它让开发者几乎不可能突然扔出一个保龄球，也让他们在轮到自己时更有可能成功击倒所有球瓶。尽管团队中

最有经验的人可能会对某件事情应该如何完成有最多意见，他们可能没有写下最佳的说明。让团队负责人和一个新手开发者一起完成这个文档。然后确保整个软件可以持续地遵循这份文档而不需要外部支持。

让人们养成持续的习惯会使人在面临高压时不会遗漏步骤。这份文档还应该包括开发者需要用来克隆仓库和提交拉取请求的命令。一旦你看到文档对于日常任务多么有用，你或许会开始审视是否可以主动使用文档的其他流程［应急响应方案（https://drupalize.me/blog/201501/being-prepared-when-everything-goes-wrong），有人赞同吗？］。

除了这些你已经习惯编写的良好提交说明外，Bitbucket 还提供了两个工具来帮助你记录工作：Wiki 页面和 issue。在本节剩余部分，你将会学到如何使用这两个工具。

11.3.1　Wiki页面中的项目文档

要想开始与他人协作，最简单的方式是将仓库访问权限授权给另一个 Bitbucket 账户。不过稍等一下！在你开始与某个新开发者之间的新关系时，应该花些时间记录你希望怎样工作。这些步骤应该记录在文档中，并且它们应该是你希望使用的步骤。幸运的是，Bitbucket 上的 Wiki 页面十分易于编辑，因此你应该考虑将文档作为起点，而不是终点。

要为你的项目启用 Wiki 页面，请执行以下操作。

(1) 找到并点击项目中的设置齿轮。
(2) 找到并点击 Wiki 设置链接。
(3) 将设置从 No wiki（无 Wiki）改为 Private wiki（私有 Wiki）（图 11-13）。

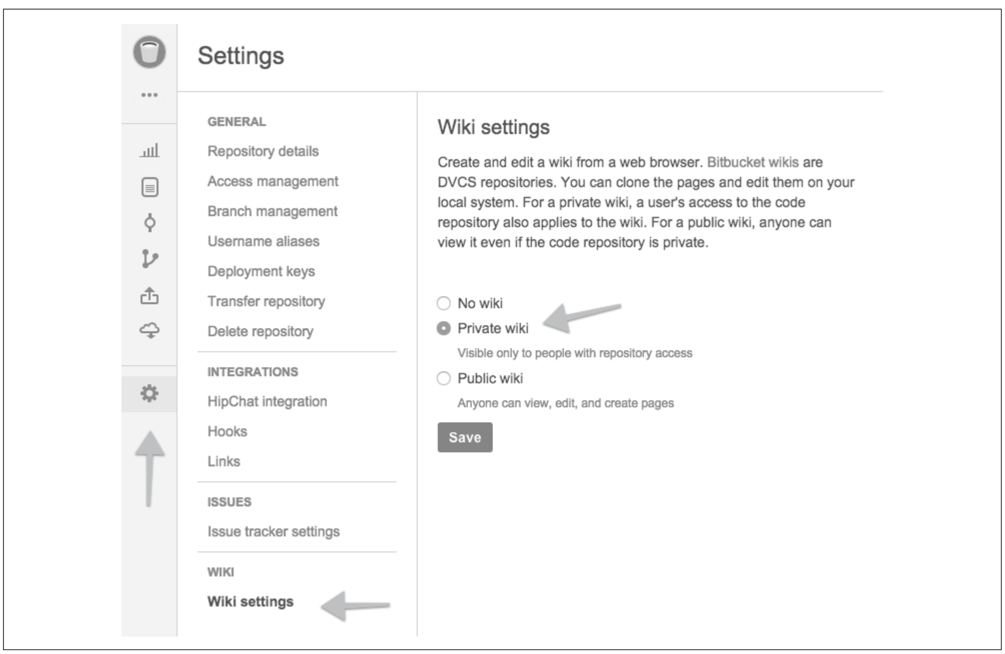


图 11-13：为你的项目启用私有 Wiki 页面

(4) 找到并点击 Save（保存）。

现在已经为你的项目启用了 Wiki 页面。在侧边栏上将会显示一个新的图标（图 11-14）。

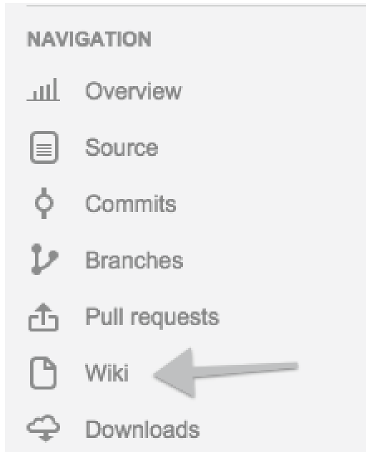


图 11-14：出现在项目侧边栏的 Wiki 图标

在 Bitbucket 中，Wiki 页面同样也是可以下载并在本地编辑的仓库。文档出现在你的 Wiki 欢迎页面（图 11-15）上。在每个 Wiki 页面的顶部有一个面包屑导航。点击项目名称之后，将会显示项目所有 Wiki 页面的列表。

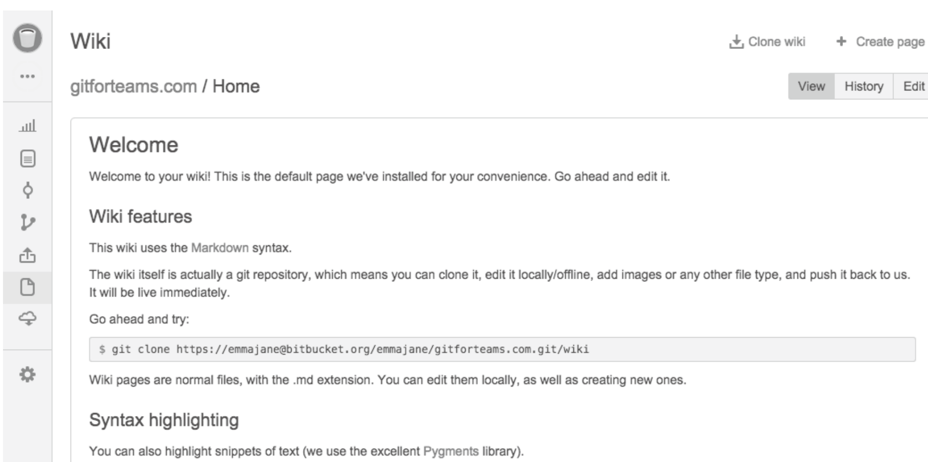


图 11-15：Bitbucket Wiki 的默认页面

Wiki 页面的编辑器是 Markdown 文件常见的工具栏（图 11-16）。

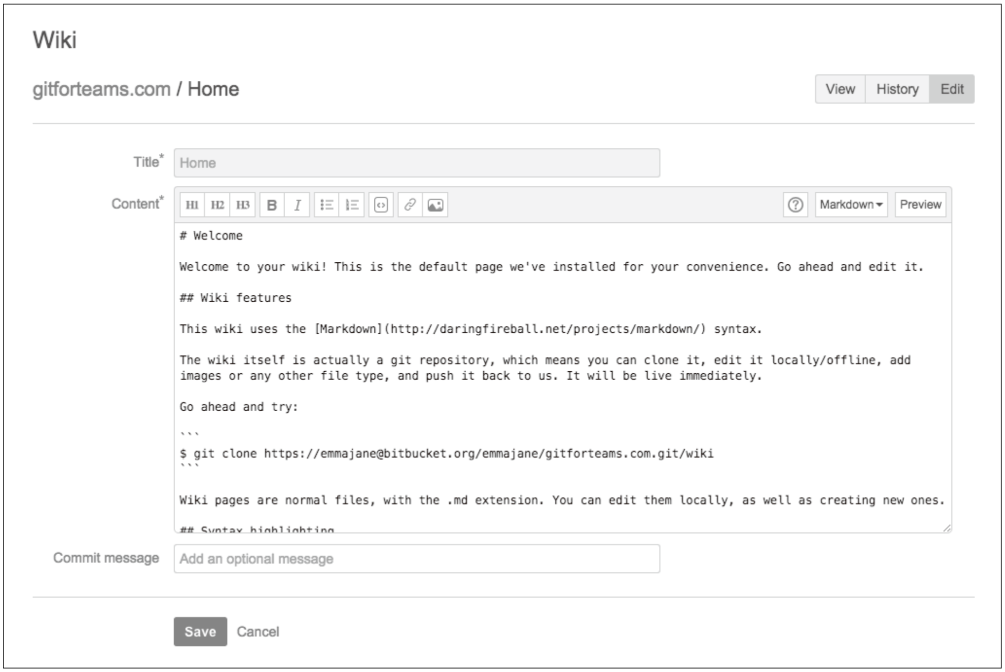


图 11-16：Wiki 页面上的 Markdown 编辑器

你至少应该为项目记录以下事项。

- 分支约定
- 向项目提交新工作的分步说明
- 同行评审的分步说明
- 部署说明，包括向谁发邮件以及邮件的复制粘贴模板

只要你认为人们有可能会出现不同的看法，或者有可能会忘记某一步，你都应该在文档中记录。它不需要很长，但必须是正确的，定期查看你的团队是否喜欢采取别的流程。有可能团队找到了一种没有记录在文档中，但更有效的方式。

11.3.2　使用issue跟踪你的更改

issue 跟踪是另一种形式的文档。尽管与 Wiki 页面相比，issue 存在时间更短，在一个工单中记录信息直接地提供了代码开发任务的商业价值，或功能构建的原因。

要启用 issue 跟踪工具，请完成以下步骤。

(1) 前往你的项目仓库。
(2) 找到并点击 Settings（设置）图标。
(3) 找到并点击 Issue tracker settings（issue 跟踪工具设置）。
(4) 将表单选项从 No issue tracker（无 issue 跟踪工具）改为 Private issue tracker（私有 issue 跟踪工具）。

(5) 可以选择输入一个新的 issue 消息。
(6) 找到并点击 Save（保存）按钮。

如你在图 11-17 中所见，我在 New issue message（新建 issue 消息）字段中为所有新 issue 提供了一条默认消息。

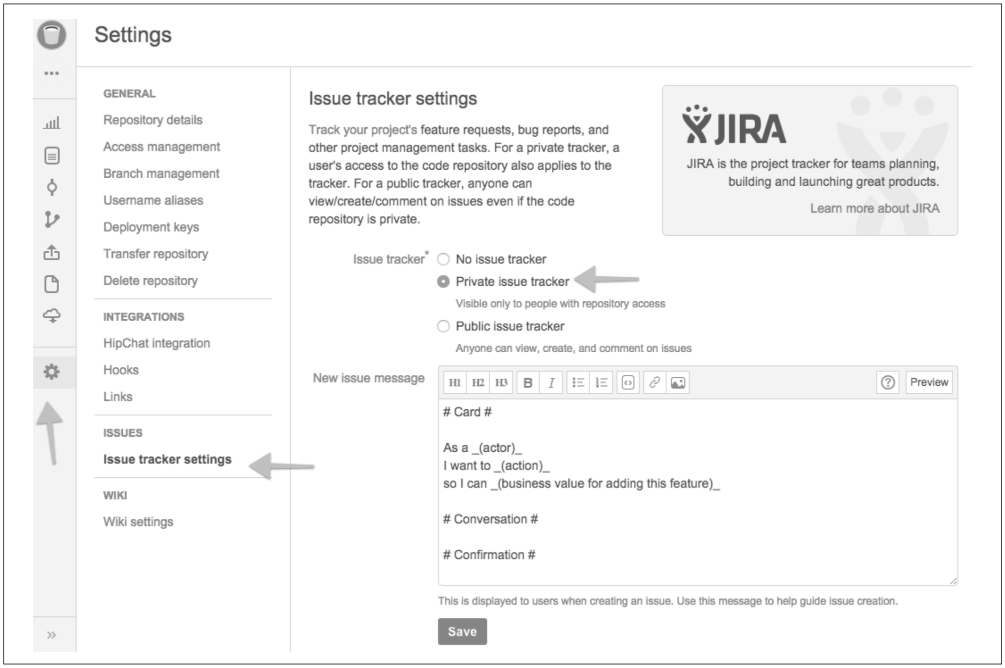


图 11-17：启用 issue 跟踪工具，并且为新 issue 添加默认消息

这条消息提醒人们遵循敏捷开发的约定，即卡片（Card）、讨论（Conversion）和确认（Confirmation）。文字将会出现在新建 issue 表单中。你的团队或许有另一组想要遵循的格式。我曾经使用过并且非常喜欢的另一种格式用的是这样的标题：QA 测试、原因和详细信息。

创建优秀 issue

确保卡片清晰地定义了谁将以何种方式受益于即将构建的功能。换句话说：商业价值是什么？这将允许进行这个任务的人询问利益干系人关于细节实现的问题。理解 issue 在更大的项目上下文中的作用将会保证整个项目使用了正确搭建的脚手架，而不是胶带。

并非所有 issue 都是作为新功能开始的。偶尔 bug 会潜入你的软件。优秀的 bug 报告应该包括：复现问题的步骤、期望的结果和上述步骤产生的实际结果，包含一张屏幕截图或者结果的视频。

关于创建优秀 issue 的信息可以在创建工单与报告 issue（http://gitforteams.com/resources/great-issues.html）中了解。

现在已经在你的项目中启用了 issue 跟踪（图 11-18）。

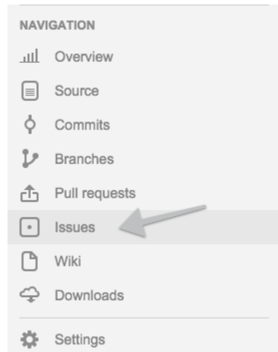


图 11-18：Issues 图标现在出现在项目侧边栏中

要创建新 issue，请完成以下步骤。

(1) 在项目侧边栏中，找到并点击 Issues 图标。
(2) 如果这是你第一次使用 issue 跟踪工具，那么将会显示一个欢迎页面，点击 Create your first issue（创建第一个 issue）以继续。如果这不是你的第一次，那么将会显示所有 issue 的摘要页面。在这个页面中，找到并点击 Create Issue（创建 issue）按钮。此时将会显示一个 issue 创建表单。
(3) 在新 issue 创建表单中（图 11-19），为你的 issue 添加一个标题和描述。Assignee（接受者）、Kind（类型）和 Priority（优先级）的默认值或许足够。

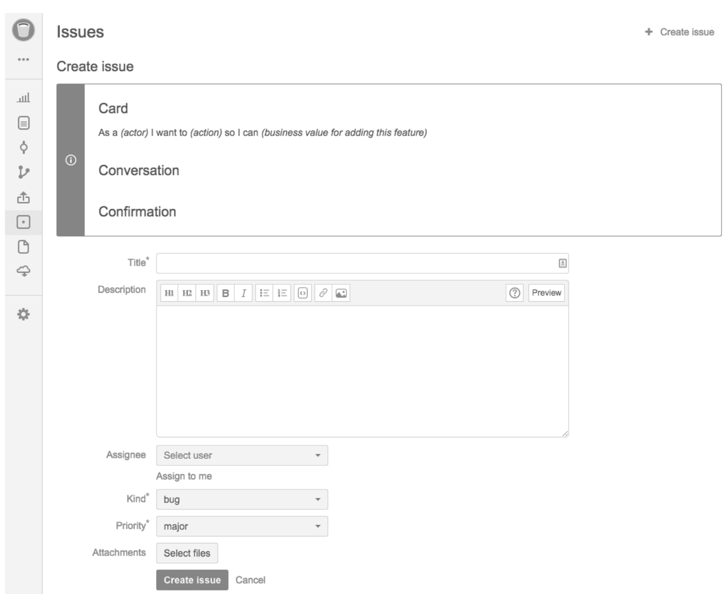


图 11-19：新 issue 创建表单

(4) 当你尽己所能描述新 issue 后，点击 Create issue 按钮。

你的 issue 已经创建（图 11-20）并可从项目侧边栏的 Issues 图标访问，你现在可以等待有人开始解决这个 issue。不过，首先你需要授权他们可以访问项目，这样就不需要自己解决每个工单。

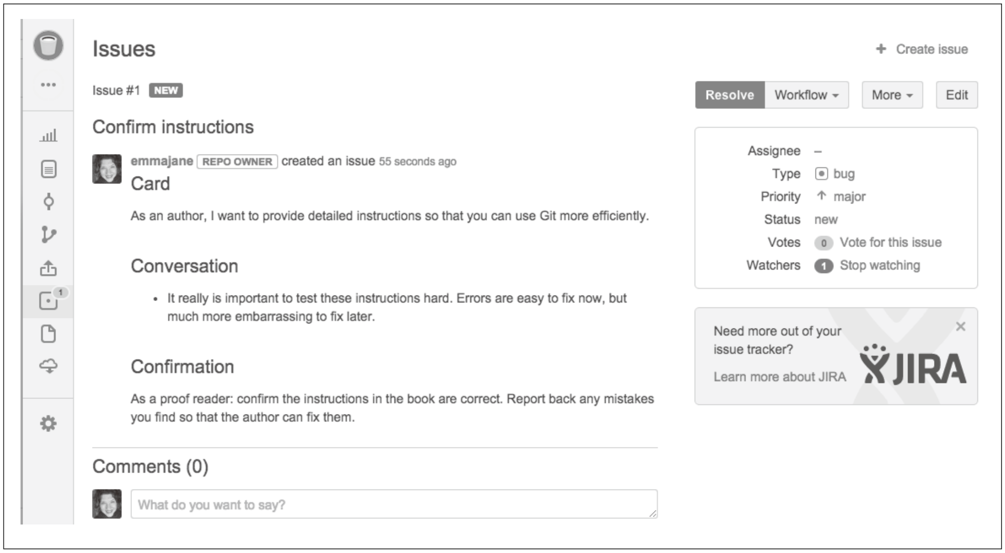


图 11-20：issue 摘要页面

11.4 访问控制

尽管没有统计数据表明这是使用 Bitbucket 最流行的方式，但我看到的团队中最常见的做法是保留默认值：私有仓库，且允许私有派生。对于小团队来说，我最常看到的工作流是开发者创建他们自己的派生，从他们自己的仓库中提交拉取请求（图 11-21）。然而，只有一两个人的团队通常省略为团队中每个人创建单独仓库这一步，而是直接在主仓库中协作（图 11-22）。

每个开发者拥有一个单独的仓库并没有禁止人们向主项目仓库做出贡献，如果你正在实施同行评审，事实上正如你所愿：每个开发者都可以向主项目仓库进行提交，但约定只限制他们在经过评审前不能提交自己的工作。然而，如果你正在与质量保证团队合作，则会希望进行限制，只有 QA 团队有主项目仓库的写入权限。在这种情况下，每个开发者需要创建一份项目的派生才能提交他们的工作。

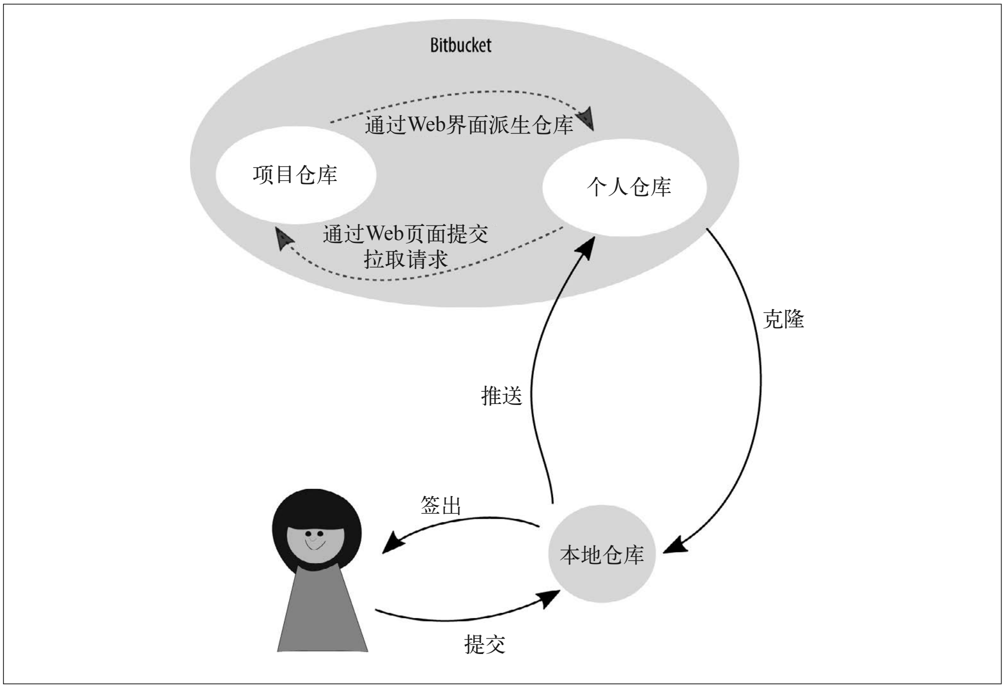


图 11-21：多人团队通常使用 Bitbucket 上的一个中间仓库

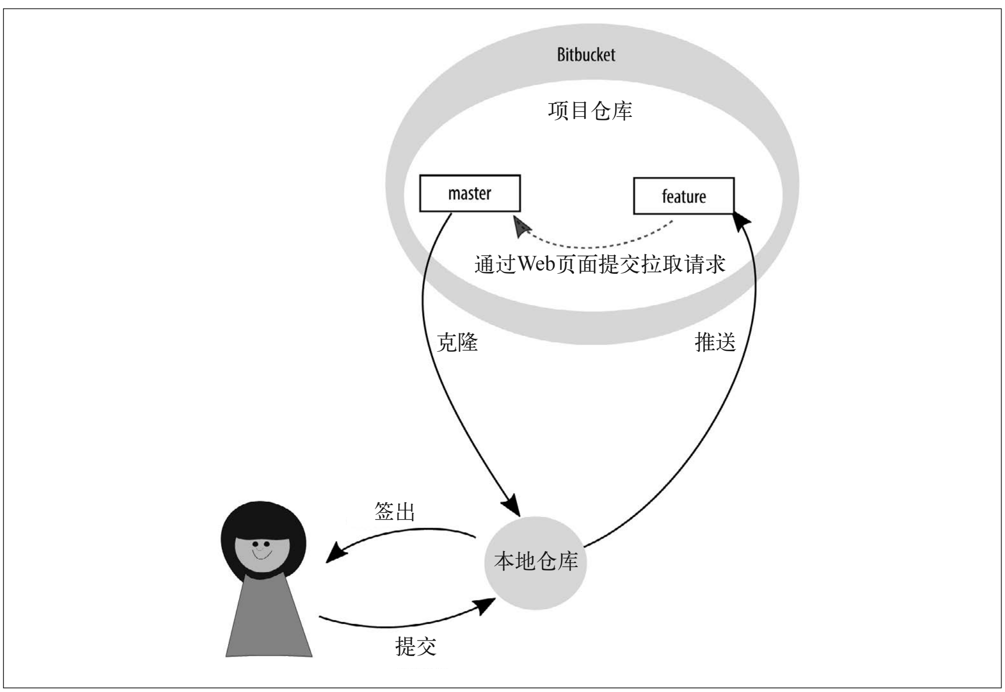


图 11-22：只有一两个人的团队通常直接在共享仓库中工作

11.4.1 共享权限

如果你的团队中都是十分可信的开发者，你可以选择让他们提交到同一个仓库，并约定每个分支的用途。

要授权开发者访问你的仓库，请完成以下步骤。

(1) 前往 Settings（设置）→ Access management（访问管理）。
(2) 在标签为 Users（用户）的字段中添加你想要添加的开发者的 Bitbucket 用户名或电子邮件地址。
(3) 将访问层级从 read（只读）改为 write（读写）。
(4) 点击 Add（添加）。

为你想要将仓库与之共享的每个开发者重复这些步骤。开发者将能够做到除了项目管理之外的所有事情。你已经准备好了文档，对吧？因为现在唯一维系这个项目的事情就是你文档中记录并自己同意严格遵守的社区约定。

11.4.2 每个开发者分别派生项目

随着你的团队增长，你可能希望避免某些团队可以直接拥有仓库的写入权限。或许你希望仅 QA 团队可以写入主仓库。在这种情况下，开发者需要创建一个项目的派生，并通过拉取请求提交他们的工作。

要创建项目的派生，请完成以下步骤。

(1) 找到并点击项目侧边栏上的 Actions（操作）图标，即图标下方的三个点。
(2) 点击标签为 Fork（派生）的链接。
(3) 此时将会显示项目创建页面，与你第一次创建自己的 Bitbucket 仓库时所见到的十分类似。在这个表单中，你可以保持所有默认值不变。
(4) 你可以禁用 Wiki 和 Issues 选项。你应该在主项目仓库中跟踪这些信息。
(5) 为了完成这个流程，点击 Fork repository（派生仓库）。

现在，你可以创建一份本地克隆并开始你的工作，步骤如下。

(1) 点击项目侧边栏上的 Actions（操作）图标。
(2) 选择 Clone（克隆）。此时将会显示一个模式窗口。
(3) 在模式窗口中，选择并复制命令行指令。
(4) 在命令行中，前往你想要放置克隆的仓库副本的目录。
(5) 粘贴 Bitbucket 提供的命令。这个仓库将会开始下载。

一旦仓库下载完成后，你可以创建一个新的分支并开始解决你的工单。

11.4.3 通过保护分支限制访问

如果你之前使用过 Subversion，那么在转到 Git 并发现它几乎没有访问控制时，可能会感到很惊讶。尤其是内置这个功能。Git 内置了让你通过钩子构建访问控制的能力。这些钩子允许你在提交发生前后或者推送到远程仓库前后使用脚本来编写回应。如果你自己托管 Git 仓

库，那么可以考虑利用这些钩子，但如果你习惯于使用代码托管系统，则可能尚未听说过这个功能。（甚至即使你听说过，假如正在学习 Git 的基础，也不会想到去编写钩子。）

幸运的是，Bitbucket 帮你完成了这些工作。你可以在 Web 界面上授权每个人或每个团队的写入权限。在第 2 章和第 3 章中，你和团队一起制定了治理策略，或许还有分支策略。我不会再次介绍这些。如果你不确定你应该如何利用这些访问控制选项，应该回去复习这些章节。

你之前学到了如何授权访问整个仓库。在本节中，你将会学到如何授权每个分支的访问。在继续本节之前，确保你已经授予想要一起协作的开发者访问仓库的权限。

要限制分支访问，请完成以下步骤。

(1) 前往 Settings（设置）→ Branch management（分支管理）。
(2) 在标题 limit pushes to specific users and groups（限制特定用户和群组的推送）下的第一个字段中，输入你想要限制访问的分支名称；在第二个字段中，输入你允许更新该分支中文件的成员名称。
(3) 点击 Add（添加）按钮。

除了列出的成员外，其他成员都被限制了将代码推送到该分支的权限。图 11-23 显示，一旦你添加某个成员之后，就可以继续添加下一个。

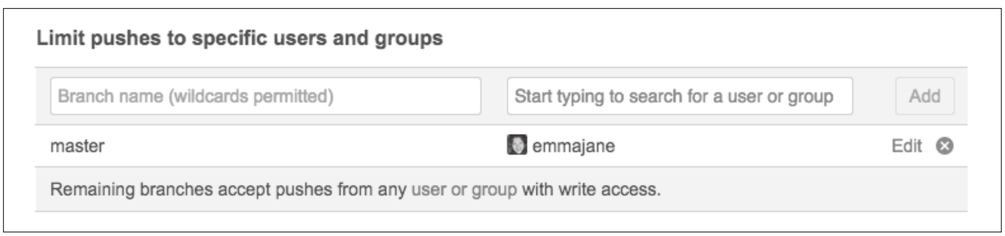


图 11-23：阻止其他人将代码推送到某个分支

在同一个设置页面，Bitbucket 同样提供了阻止删除任何分支的选项。尽管当你的团队在指导如何安全地使用 Git 工作时，这两个选项没有那么吸引人，但你的“朋友”可能仍然需要它。[没关系，我理解，Atlassian（https://www.atlassian.com/）也是，这也是为什么它为你提供了这两个功能。]

一旦设置完成后，如果有人试图执行一个受限的操作，则会返回到一个错误页面。例 11-3 展示了一个例子，当我尝试删除一个名为 `master` 的受保护分支时会发生什么。

例 11-3　在删除锁定分支时的错误

```
$ git push bitbucket master --delete
remote: permission denied to delete branch master
To git@bitbucket.org:emmajane/gitforteams.git
 ! [remote rejected] locked (pre-receive hook declined)
error: failed to push some refs to 'git@bitbucket.org:emmajane/gitforteams.git'
```

如果你决定使用访问控制，确保你已清楚地与你的团队沟通过这些限制。在开发者弄不明白他们为什么无法将代码推送到项目仓库时，这将帮助他们避免陷入必然的沮丧。你不需

要提供没人会看的长篇大论，但确实需要告诉他们做出这些决定的原因，以及让你的系统成为独一无二的雪花状架构的技巧。

更多关于分支管理（https://confluence.atlassian.com/display/BITBUCKET/Branch+management）的信息可以在 Bitbucket 上找到。你可能还对 Atlassian 的 Git 钩子（https://bitly.com/gits-hooks/）使用概述感兴趣。

11.5 拉取请求

你的开发者要想将工作添加回仓库，他们需要权限。如果没有权限（要么按照社区约定完成同行评审，要么通过强制的访问控制），开发者需要创建一个拉取请求来考虑将他们的工作添加到主项目中。

Atlassian 的 Bitbucket 官方文档写得非常好。使用拉取请求（https://confluence.atlassian.com/bitbucket/work-with-pull-requests-223220593.html）介绍了一些额外的功能，并会一直保持最新（如果本节中使用的指令已过时）。

11.5.1 提交拉取请求

在你的工单分支中完成 issue 相关的工作，并将代码推送至服务器之后，你可以发起一个拉取请求来将你的工作整合到主项目仓库。这个界面选项会根据你选择的访问控制方法不同而略有差异。不过，基本流程如下所示。

(1) 找到并点击侧边栏上的 Pull requests（拉取请求）图标。
(2) 找到并点击 Create pull request（创建拉取请求）链接。针对你的请求的一个新表单将会出现（图 11-24）。

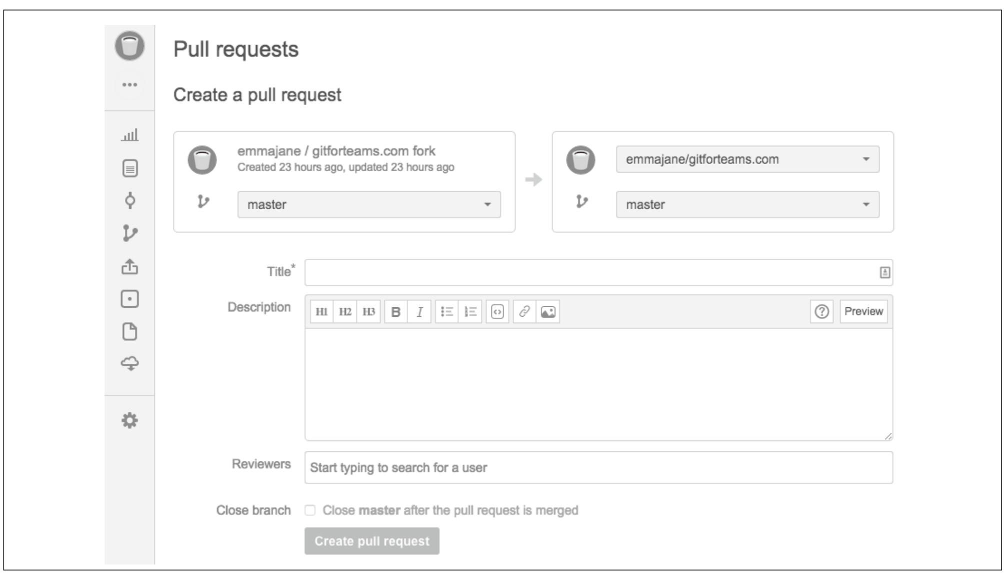


图 11-24：拉取请求新建表单

(3) 你的当前仓库位于左侧。在选项中，选择包含你想要并入主项目的更改的分支。
(4) 目标分支位于右侧。如果你的仓库是派生的，你可以选择目标仓库和目标分支。
(5) 为你的拉取请求添加标题和描述。理想情况下，你的描述应该引用你打算关闭的 issue。
(6) 如果你希望某人评审你的工作，可以在拉取请求中输入他的名字。
(7) 你可以选择让 Bitbucket 替你进行一些维护，在拉取请求被接受且工单被关闭后，删除工单分支。
(8) 最后，当表单完成后，点击 Create pull request 按钮。

作为开发者，你现在必须等到你的工作被评审并接受后进入项目，或者被要求更新你的请求。

11.5.2　接受拉取请求

一旦拉取请求提交后，评审者会决定提议的更改是否值得加入主分支。第 8 章详细介绍了评审环节。拉取请求的摘要页面允许评审者评论被提议的工作。讨论可能会导致拉取请求更新，或确认工作已经正确地完成并可以并入项目。

假设没有冲突，你可以通过点击请求中的 Merge（合并）按钮来接受拉取请求。

然而，如果存在合并冲突，这个过程会更复杂一些。通常，解决冲突的最佳人选是添加的新代码的开发者。通常代码在等待评审时会变得过时。让开发者更新她的工单分支，使分支包含父分支（或源分支）中最新的更改，如下所示。

```
$ git pull --rebase=preserve
```

如果提交这个拉取请求的开发者无法解决合并冲突，你可能需要自己完成这一步。幸运的是，Bitbucket 为你提供了可以复制粘贴的命令来下载工单分支并解决冲突。

11.6　使用Atlassian Connect扩展Bitbucket

除了 Bitbucket 自带的这些功能之外，插件 API Atlassian Connect 提供了一个免费和付费插件的市场。

要找到与你项目相关的插件，请完成以下步骤。

(1) 点击页面右上角的用户图标，前往你的项目管理页面，然后选择 Manage account（管理账户）。
(2) 从你的账户的左边栏中，找到并点击 Find new add-ons（寻找新插件）。所有插件的列表将会出现在主要区域中（图 11-25）。

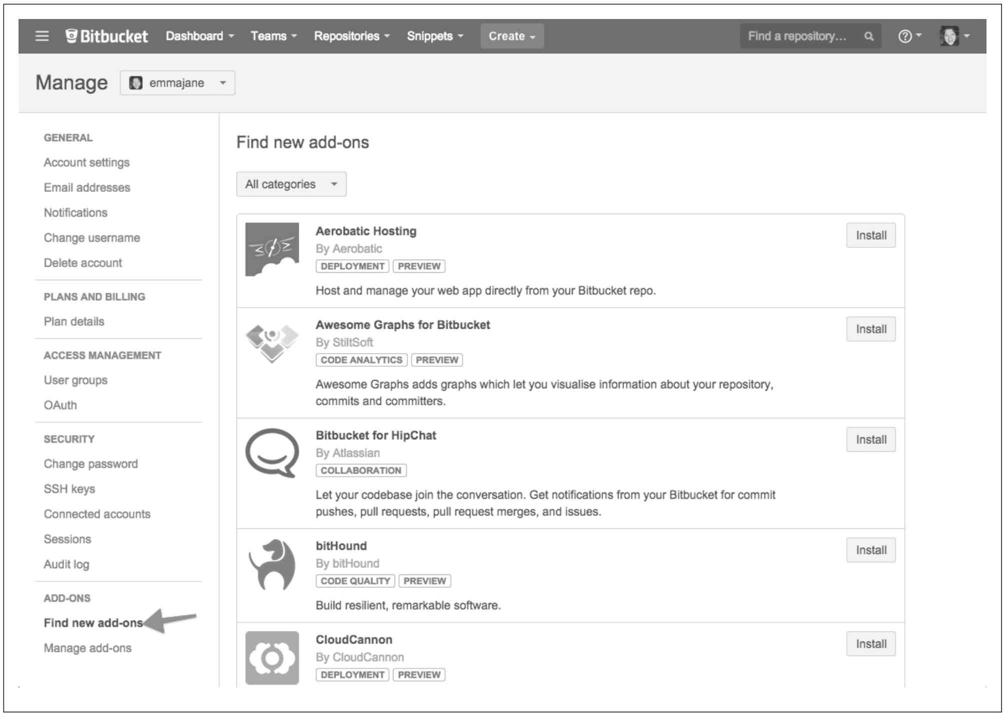


图 11-25：通过 Atlassian Connect 可得到的可用插件列表

你可以通过类别过滤这个列表。例如：代码分析、代码质量、协作和部署。这是一个新的服务，因此当你阅读本书的时候，应该已经有很多可供浏览的插件了。下面是一些值得研究的插件。

- bitHound（https://www.bithound.io/）
  根据代码指令、可维护性和稳定性给你的 JavaScript 项目评分。对闭源项目收费，对开源项目免费。

- Aerobatic Hosting（https://www.aerobatic.com/）
  允许你部署静态网站，类似 GitHub Pages，除了它是从私有的 Bitbucket 仓库中创建的之外。

- Pull Request Auto Reviewers（https://marketplace.atlassian.com/plugins/com.atlassian.labs.pr-auto- reviewer/versions）
  允许你自动为特定类型的拉取请求分配评审者。

除了 Connect 的插件外，你还可以从自定义 URL 安装插件。你可以在 Atlassian 的开发者门户中学到更多关于开发 Connect（https://developer.atlassian.com/static/bitbucket/）的信息。有可能你的插件不仅对你的团队有用，也对其他团队有用。在你构建它的时候，考虑程序的抽象性，从而可以在市场上共享给（或卖给）其他人。

11.7　小结

在本章中，你学到了如何使用 Atlassian 的流行代码托管平台 Bitbucket。你学到了如何设置一个个人项目，并将你的仓库共享给别人。为了顺利地在私有项目中协作，你需要牢记在本章中学到的以下几项内容。

- 通过首先创建一个个人的私有仓库来熟悉你的工具。
- 通过创建可以很容易从项目仓库访问的优秀上手文档，迎接团队新成员的加入。
- 在仓库中使用基于 issue 的更新，描述在仓库中创建新分支之前在 issue 中所有提议的更改。
- 让关于访问控制的决定清晰透明。如果你限制了访问，请在文档中记录你作出这个决定的原因。

多年来我一直对 Atlassian 这家公司有很深的印象。它不断地提供易于理解、组织良好的文档和乐于助人的员工。偶尔出现问题时，它也以一种成熟的、令人尊敬的方式承担起了责任。干得漂亮，Atlassian！

第 12 章

GitLab上自行管理的协作

GitLab 是一个用于仓库管理的开源代码托管系统。它允许你跟踪仓库中的 issue，实施代码评审以及通过 Wiki 页面创建辅助的项目文档；换句话说，它与 GitHub 和 Bitbucket 几乎相同。GitLab 特有的优点是，它是一个开源产品，你不需要支付版权费用即可在任何地方安装这个软件，而且可以直接扩展这个软件，不会因受到限制而不能通过 API 和其他钩子来创建插件。

完成本章学习之后，你将具备以下技能。

- 找到所有安装相关的指令来完成设置
- 创建新项目、新用户和新群组
- 配置项目的访问控制
- 建立项目间的里程碑

本章主要介绍你作为 GitLab 实例的管理员能够执行的特殊任务，而不只是作为项目负责人使用软件。

12.1 入门

如果你一开始就遵循了本书中的活动，那么已经创建了一个仓库，并且在 GitLab 的公开版本 GitLab.com 中熟悉了 GitLab 的仓库。（如果你需要回顾一下，可以查看第 5 章中介绍的作为单人团队使用 GitLab 的说明。）

12.1.1 安装GitLab

为了利用本章剩余部分中介绍的管理功能，你应该创建一份本地 GitLab 实例，以管理账

户持有人身份登录。本章介绍的是社区版，而不是 GitLab 的企业版。企业版需要付费，并提供了额外的功能，例如 JIRA 集成。你可以在功能对比清单（https://about.gitlab.com/features/#compare）中查看两者的差异。

安装 GitLab 的推荐方式是使用 Omnibus 安装包（https://about.gitlab.com/downloads/）。这个包可以直接下载并放置于一台 Linux 服务器上，或通过一些配置服务一键安装部署。

DigitalOcean 提供 GitLab 的一键安装包（https://www.digitalocean.com/products/one-click-apps/gitlab/）。这个安装包使用 Omnibus 的 GitLab 安装程序，这意味着如果存在新功能或漏洞发布，你可以轻松地升级 GitLab。在本书编写时，DigitalOcean 是唯一一家提供 Omnibus 安装包的一键安装程序的服务商。Bitnami 和 AWS 市场只提供从源安装包部署，在部署后无法升级。仔细阅读说明，确保你不会被某个特定版本的安装绑定。

为了避免使用 GitLab 时的托管费用，你还可以在本地使用虚拟机安装。（没有听上去那么可怕。）VirtualBox 和 Vagrant 是在 Windows 和 OS X 机器上配置 Linux 服务器最简单的方法。Vagrant 的书面文档非常优秀，但如果你倾向于通过视频学习，可以参考我收集的一些稍旧版本的 Vagrant 系列视频（https://drupalize.me/videos/why-vagrant）。基本用法没有变化。

一般来说，步骤如下。

(1) 安装 Virtualbox（https://www.virtualbox.org/wiki/Downloads）。
(2) 安装 Vagrant（http://www.vagrantup.com/downloads）。

如果你正在使用 OS X，已经有一个 Virtualbox 和 Vagrant 的 `brew` 范例，那么可以直接使用。

当这两个安装包安装后，你现在可以在你的本地机器上运行一个 Ubuntu 服务器。不过，虚拟机不会预安装 GitLab。此时，你可以通过之前提到的 Ominbus 安装包来安装 GitLab，但我发现下面的 GitLab 安装程序（https://github.com/tuminoid/gitlab-installer）非常易于使用。

在命令行中完成以下步骤。

(1) 从 GitHub 上克隆安装项目，如下所示。

```
$ git clone https://github.com/tuminoid/gitlab-installer.git
```

(2) 在仓库内，将 Ruby 配置文件的名称从 gitlab.rb.example 改为 gitlab.rb。
(3) 启动虚拟机，如下所示。

```
$ vagrant up
```

新的虚拟机将准备好。用户名和密码将会在 Vagrant 启动消息的结尾输出。如果你记不住它，或者已经关闭了该窗口，也可以从安装脚本结尾（https://github.com/tuminoid/gitlab-installer/blob/master/install-gitlab.sh）找到该信息。

从源代码安装

如果你真的希望从源代码安装 GitLab，在安装指南（https://about.gitlab.com/installation/）中有关于如何继续的说明。强烈建议不要这么做，因为 GitLab 在每月 22 号会发布一个软件的新版本。使用安装包会使更新你的 GitLab 实例容易得多。

不论你选择哪种安装方法，都需要以管理员身份登录你的新 GitLab 实例来学习本章剩余内容。一旦你登录之后，将会出现图 12-1 中所示的欢迎页面。

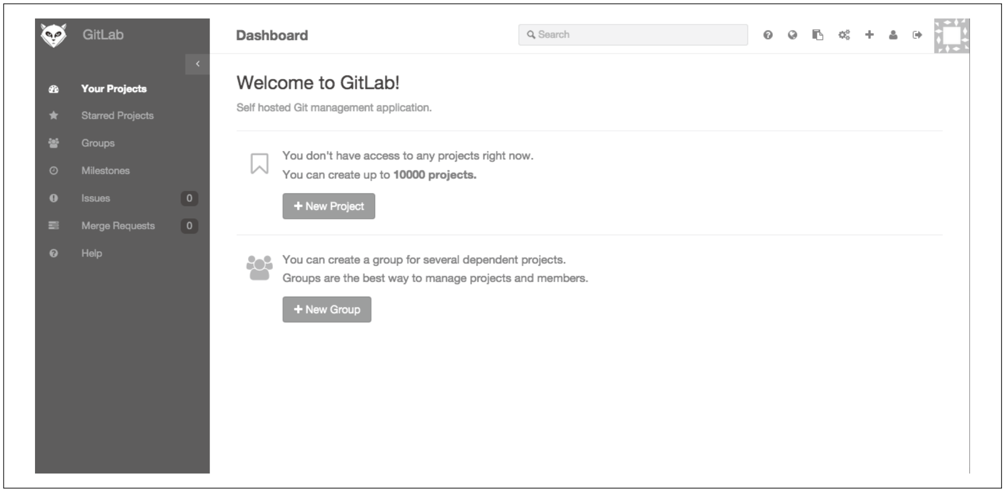


图 12-1：GitLab 的欢迎页面

如果你无法完成安装，我鼓励你快速浏览本章剩余内容，看看你能够获得什么好处，确认它是否值得你完成安装。

12.1.2　设置管理账户

你或许会选择保留一个通用的管理账户，或在与团队一起开发软件时将它作为你自己的账户。出于习惯，我倾向于创建一个拥有较少权限的账户用于日常使用，并维护根账户，用于安装新插件、升级软件等任务以及其他管理性任务。

要配置你的账户，请完成以下步骤。

(1) 在顶部菜单中，找到并点击个人资料设置图标（人像）。
(2) 在左侧边栏中，检查个人资料设置页面中的每一项。

- 个人资料
  你的姓名和公开的详细信息，如 Skype 和 Twitter。

- 账户
  私有令牌、双重身份验证、用户名和删除账户的能力。

- 应用
  管理将 GitLab 作为 OAuth 提供者的应用以及你已授权使用你的账户的应用。

- 电子邮件
  主要电子邮件（头像、提交积分）、通知电子邮件和公开电子邮件（显示的电子邮件）。其中任意一个地址可以用于连接你的提交。

- 密码
  重设你的密码。
- 通知
  你的通知电子邮件以及你的通知层级。默认情况下，你将会受到相关资源（你的提交和资产等）的通知。你也可以选择 Watch（给定项目的所有通知）、Mention（只有在 issue 或评论中被 @ 到时）或 Disabled（不接受任何通知）。
- SSH 密钥
  注意：如果你通过 SSH 密钥登录到账户中，那么将无法通过 SSH 连接仓库。此时将会出现提醒，直到提醒被关闭或你上传了 SSH 密钥为止。
- 设计
  侧边栏的配色设置或代码语法高亮。
- 历史
  该账户创造的所有活动。包括你进行的操作，例如提交、创建新项目等。

一旦你配置好了管理账户，就可以继续下一步。如果你决定立刻创建另一个账户，请跳到 12.3 节。

12.1.3 管理信息中心

以管理用户身份登录后，你将能够访问一些额外的页面和功能，而公开的 GitLab.com 网站上的非管理员则无法访问。大多数功能可以在 Admin area（管理区域）中找到。

在顶部菜单中，点击标签为 Admin area（管理区域）的齿轮图标。此时将会出现图 12-2 中所示的页面。

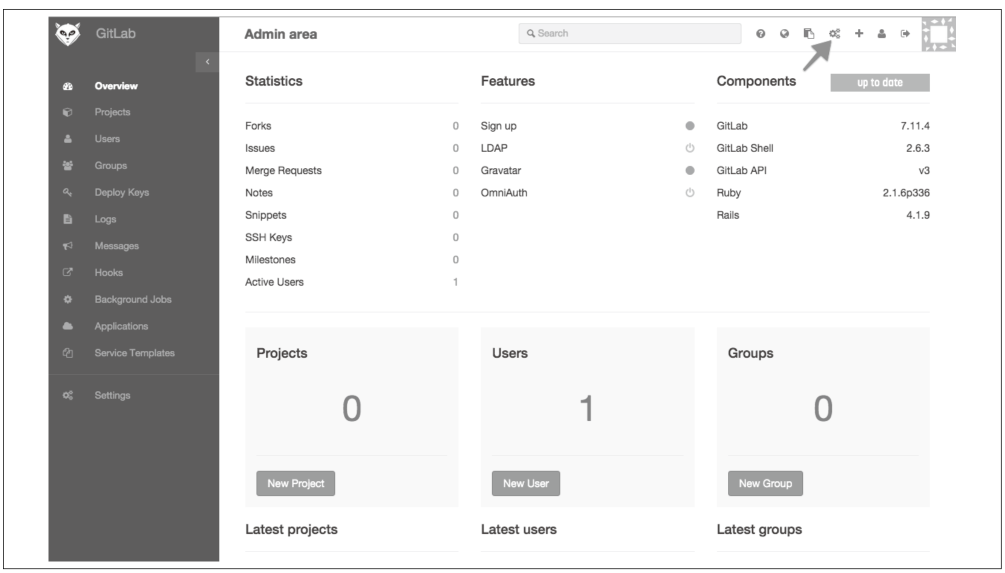


图 12-2：管理信息中心包括项目详细信息摘要和 GitLab 安装版本的状态报表

这个页面提供了该 GitLab 实例安装的组件摘要，包括你安装的 GitLab、GitLab 命令行、GitLab API、Ruby 和 Rails 的软件版本。还有一列可用的功能，并用状态指示灯显示哪些功能已启用。在图 12-2 中，你可以看到 Sign up（注册）和 Gravatar 被启用，而 LDAP 和 OmniAuth 被禁用。封闭的圆圈是绿色的，代表“打开”；而“关闭”符号是代表待机的图标。不幸的是，这些图标是通过 CSS 本身提供的，目前不提供等价的文本描述。

以下是左侧边栏中的选项，其中每个选项的含义都一目了然。

- Overview（概览）
  即你在图 12-2 中看到的页面。提供站点的快速状态概览，以及创建新用户、新项目和新群组的快速链接。

- Projects（项目）
  在站点内搜索项目，提供按每个用户、不活跃（最近 6 个月无活动）和可见层级（私有、内部或公开）的筛选。

- Users（用户）
  在站点内搜索账户，提供按管理员、封禁账户和无项目用户的筛选。

- Groups（群组）
  按名称搜索群组，或添加一个新群组。此页面不提供筛选。

- Deploy keys（部署密钥）
  所有用于部署的密钥列表，你也可以在此页面中添加新的密钥。

- Logs（日志）
  githost.log、application.log、production.log 和 sidekiq.log 中的每一个的最后 2000 行。

- Messages（消息）
  向所有系统账户添加定时广播消息的能力。可用于计划维护、近期升级等。

- System hooks（系统钩子）
  所有现有系统钩子的列表。在钩子列表中，你可以测试一个钩子或移除它，还可以在此页面上添加新的钩子（URL）。

- Background jobs（后台任务）
  在 sidekiq（http://sidekiq.org/）中运行的后台任务摘要。

- System OAuth applications（系统 OAuth 应用）
  现有应用的列表和添加新应用的能力（标题和跳转 URI 字段）。

- Service templates（服务模板）
  服务模板允许你设置项目服务的默认值。不同的服务可以选择不同的设置项。例如，如 Asana 和 Buildkite 这样的外部服务有 API 密钥字段。如 JIRA、Redmine 这样的服务还有配置字段［Project URL（项目 URL）、Issues URL、New Issue URL（新建 issue URL）］。如 Email 推送等服务，还有触发的开关（推送事件或标签推送事件）。如果你希望整合第三方服务，可以快速浏览这个页面。

- Application settings（应用设置）
  包括默认的项目设置和全站配置选项。

为了锁定你的 GitLab 实例，你需要使用 Application settings（应用设置）页面上的不同选项。GitLab 上的设置较为自由。默认情况下，应用对新注册用户开放，后者受限至多拥有 10 个仓库，但新仓库的默认设置是私有的。

Features（功能）栏包括以下设置（默认均启用）。

- Signup（注册）
  允许人们创建账户。

- Signin（登录）
  允许人们进行验证。如果你想要一个只读的公开仓库，它将是适合你的选项。

- Gravatar（通用头像服务）
  用户个人资料图片的集成（需要网络连接）。

- Twitter
  向用户提供按钮来将新建的公开或内部项目分享到 Twitter 上。

- Version check enabled（启用版本检查）
  检查是否存在更新版本的 GitLab。

Visibility and access control（可见性和访问控制）包括以下设置。

- Default branch protection（默认分支保护）
  选项包括以下几个。
  - Not protected（不受保护，开发者和 Master 可以推送提交、删除分支以及强制推送新的提交）
  - Partially protected（部分保护，开发者可以推送新的提交，Master 可以进行任何更改）
  - Fully protected（完全保护，只有 Master 可以在仓库中进行更改）。
    默认情况下，Fully protected 被启用。
- Default project visibility（默认项目可见性）
  选项包括以下几个。
  - Private（私有，为每个用户明确授权项目访问）
  - Internal（内部，项目可以被任何登录用户克隆）
  - Public（公开，项目不需要任何验证即可被克隆）。
    默认情况下，Private 被启用。
- Default snippet visibility（默认便签可见性）
  选项包括 Private（私有）、Internal（内部）和 Public（公开）。默认情况下，Private 被启用。

- Restricted visibility levels（受限可见性等级）
  选中的等级无法用于项目或便签的非管理员用户。

- Restricted domains for sign-ups（受限注册域名）
  只允许拥有指定域名的电子邮件账户的用户创建账户。可使用通配符。

还有以下这些限制设置。

- Default projects limit（默认项目限制）
  默认情况下，每个账户允许拥有 10 个仓库，包括开发者需要向项目提交合并提交的私有派生。如果开发者同时进行多个内部项目，这个数字可能需要增加。
- Maximum attachment size（附件大小限制，MB）
  默认情况下，这个值被设为 10MB。对大多数屏幕截图来说应该已经足够，但如果你还想要在 issue 后附加设计资源，这个值可能不够大。

最后，还有以下这些登录限制。

- Home page URL（首页 URL）
  未验证用户在访问除了注册页面以外的任何页面时，应该被重定向到 URL。如果未设置，人们将会被重定向到登录页面。
- Sign in text（登录文本）
  这个文本出现在登录页面中 GitLab 的描述下方。你应该用一个标题分离你的文本和 GitLab 的描述。

通过配置每个选项，你可以为你的 GitLab 实例创建一个合适的起点。例如，如果你希望让它仅为批准的人员进行官方工作提供服务，则可以调整以下这些选项。

- 禁用注册功能。
- 禁用 Twitter 功能（从界面上移除鼓励发推文的按钮）
- 设置 Restricted visibility levels（受限可见性等级），禁用公开仓库和便签（查看所有仓库需要登录）。

如果你希望让你的实例更加开放一些，则可能会像下面这样调整设置。

- 启用注册功能。
- 禁用 Twitter 功能。
- 禁用非管理员的公开仓库。
- 限制域名，只有你的组织成员可以注册。

除了这些设置外，你可以进一步自定义每个项目的设置来满足你的需求。

12.2 项目

你的组织或许已经有了一些代码项目，可能使用或没有使用 Git 进行版本管理。为了在 GitLab 中开始你的设置流程，你或许会想要从成员或项目开始。从项目开始的好处是：人们第一次登录后已有可以参与的项目。如果你和经验丰富的 Git 用户一起工作，你或许会希望授权少数人先去设置项目。

创建项目

项目事实上是一个提供 issue 队列和 Wiki 页面等支持工具的仓库。在 GitLab 中创建新项目时，你可以选择从 GitHub、BItbucket、Gitorious、Google Code 或任何其他你的 GitLab 实例可以通过 URL 访问到的仓库导入。

要创建新项目，请完成以下步骤。

(1) 在菜单顶部，找到并点击 New repository（新建仓库）图标。这个图标是一个 +。此时将会显示项目创建表单。

(2) 完成图 12-3 中新项目需要的下列每个字段。

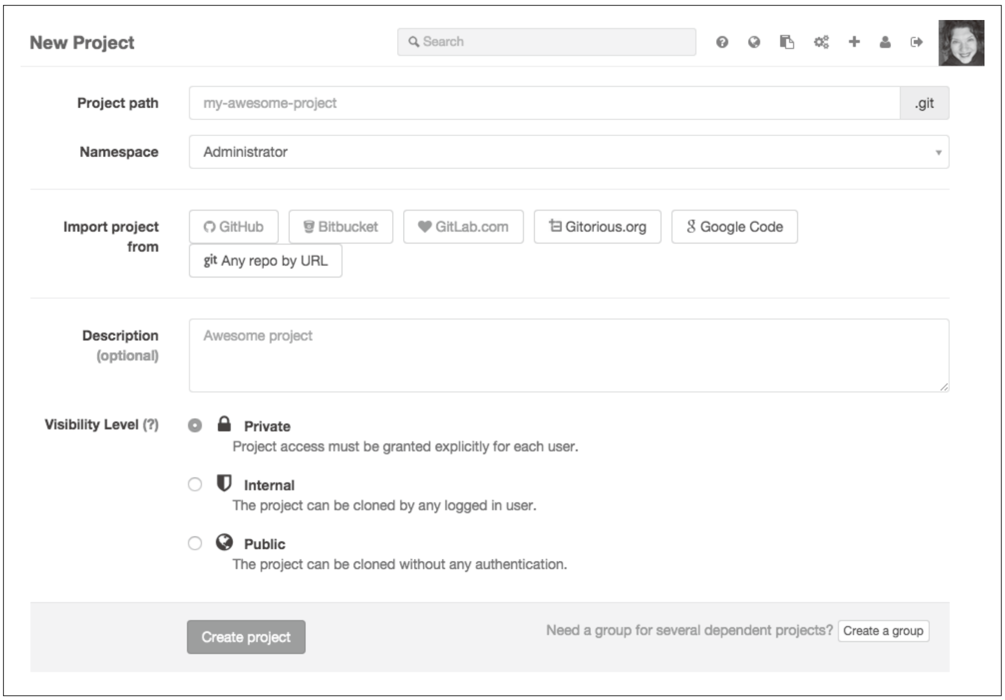


图 12-3：新项目创建表单

- Project path（项目路径）
  你的项目页面的 URL。仅可使用小写字符和连字符。

- Namespace（命名空间）
  这个项目应该归属的账户或群组名称。默认情况下，会选中你自己的账户。

- Import project from（导入项目）
  如果项目已经存在，你可以从列出的服务之一中导入。GitLab 必须能够访问那个服务以完成导入；换句话说，仓库不能位于无法联网的防火墙后，而你需要启用项目的 OAuth 访问。弹出窗口中的说明（图 12-4）会将你带到你想要连接的服务的相关文档页面。

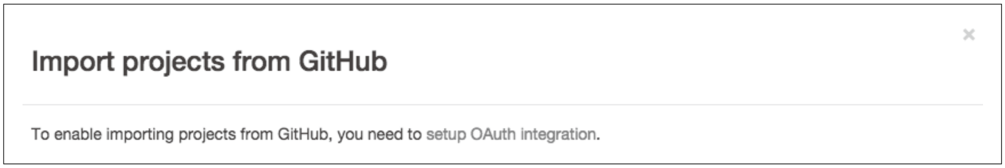


图 12-4：在点击 GitHub 按钮后，将会显示此弹出窗口，告诉你 GitLab 没有从 GitHub 中导入的权限

- Description（描述）
  用于列表中的项目信息。它不是一份完整的 README。
- Visibility Level（可见性层级）
  在 Private（私有，只有认证用户可见）、Internal（内部，所有登录用户可见）或 Public（公开，对任何网站的访问者可见）中选择。

(3) 找到并点击 Create project（创建项目）按钮。

你的新项目已经创建。如果你选择从外部服务中导入，仓库、issue 和 Wiki 页面将会被导入（如果支持）。此时将会显示新项目页面。

在项目导入后，你现在可以为项目添加管理员和开发者。

12.3 用户账户

GitLab 允许你创建特定角色的用户。这些角色可以用于调整项目的读写权限。如果你熟悉 Subversion 的分支锁定，对这些访问限制也不会陌生。在本节中，你将学到如何设置每个用户账户并为项目添加成员。

12.3.1 创建用户账户

要创建一个新用户账户，你可以从几个不同地方开始。最容易访问的是从管理区域的概览中，步骤如下。

(1) 在右上方点击标签为 Admin area（管理区域）的齿轮图标。此时将会显示管理区域的概览页面。
(2) 找到并点击 New user（新建用户）按钮。此时将会显示新用户创建表单。

这个表单，如图 12-5 所示，被分为了三个区域：Account（账户）、Access（访问）和 Profile（个人资料）。

Account（账户）部分中的字段都是必填字段，包括以下这些。

- Name（姓名）
  账户的显示名称。
- Username（用户名）
  账户的登录名。

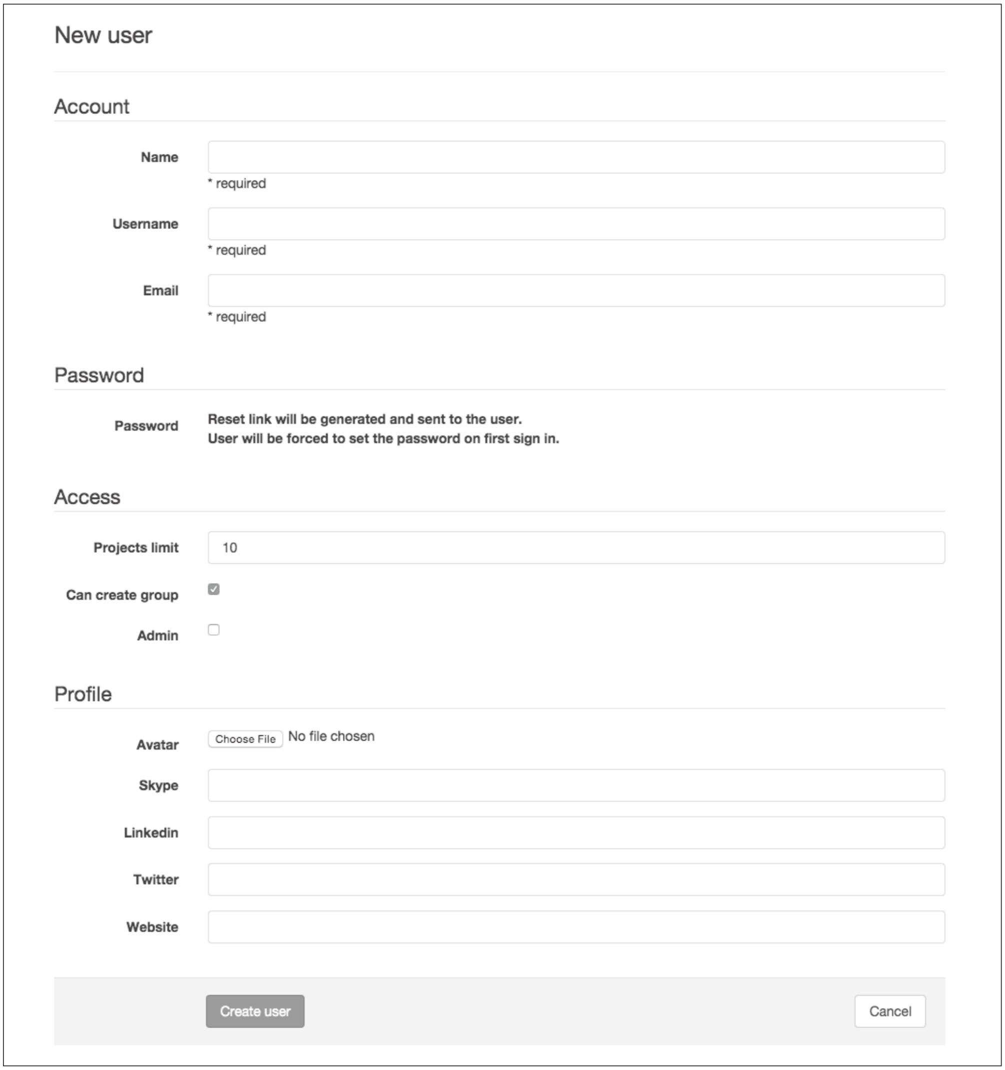


图 12-5：新用户账户创建表单被分成三个区域：Account（账户，必填字段）、Access（访问）和 Profile（个人资料）

- Email（电子邮件）
  账户的电子邮件地址。

Access（访问）部分中的字段的默认值通常是适当的，包括以下这些。

- Project limit（项目限制）
  默认数量是你之前为全站设置的值。GitLab 将默认值设为 10。
- Can create group（能够创建群组）
  聚合项目的能力。这个功能在其他系统中被为团队或组织。这个选项默认启用。

- Admin（管理员）
  允许该用户管理 GitLab 软件。这个选项默认启用。

最后，还有一个 Profile（个人资料）部分，其中包含以下几个用于上传照片和社交媒体链接的字段。

- Avatar——如果 Gravatar 已启用，则不必上传单独的用户个人资料图片
- Skype
- LinkedIn
- Twitter
- Website（网站）

尽管你可以花些时间在填写个人资料详细信息上，但不是所有员工都想要在工作系统中绑定他的社交账户。将它们留空可能会更合适。

要创建用户账户，请完成以下步骤。

(1) 填写之前描述过的 Account（账户）详细信息。
(2) 确认 Account 详细信息正确。
(3) 检查 Profile 详细信息，确保将它们留空。填写任何现在适合添加的详细信息。
(4) 找到并点击 Create user（创建用户）按钮。

新用户账户已经创建，一封通知邮件已经发送给该用户，包括一个可供用户设置密码的一次性登录链接。

除了这个手动账户创建以外，GitLAN 还集成了 LDAP（https://docs.gitlab.com/ce/administration/auth/ldap.html）和 OmniAuth（https://docs.gitlab.com/ce/integration/omniauth.html）。GitLab 的文档中介绍了这种类型的访问的设置。在本书编写时，支持 OmniAuth 的服务提供商包括 GitHub、Twitter 和 Google。

12.3.2 添加项目成员

要在项目中添加成员，请完成以下步骤。

(1) 前往项目页面。
(2) 从侧边栏中找到并点击 Settings（设置）。
(3) 在左侧边栏中，找到并点击 Members（成员）。
(4) 找到并点击 Add members（添加成员）。此时将打开一个新表单（图 12-6）。
(5) 在标签为 People（人员）的字段中，输入你想要添加到项目的成员的用户名或电子邮件。
(6) 将标签为 Project Access（项目访问）的字段调整为下列各项之一。

  - Guest（访客）
    能够查看项目、创建 issue 和留下评论
  - Reporter（报告者）
    能够克隆仓库、创建代码片段

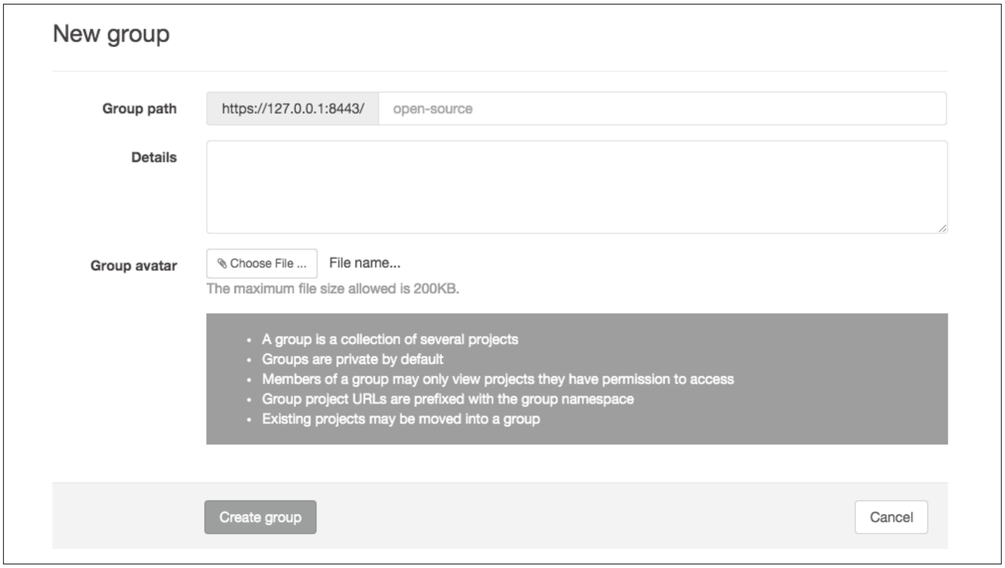


图 12-6：将用户添加到项目；搜索人员并设置合适的访问级别

- Developer（开发者）
  能够向授权的分支提交代码
- Master
  项目管理员
- Owner（所有者）
  能够移除项目

(7) 找到并点击 Add users to project（添加项目成员）。

这些账户被授予了合适的项目访问权限。如果电子邮件尚未在 GitLab 的这个示例中注册，则会发送一封邀请邮件，请求用户注册。

12.4　群组

为了收集项目，你可以使用群组。你可以将群组视为一个部门、组织或（包含子项目的）软件项目。默认情况下，群组是私有的，只有该群组的成员可以查看群组中的项目。

每个人都可以创建群组

默认情况下，所有在 GitLab 上有账户的人都有权限创建一个群组，你可以在账户创建后或从账户的设置页面，禁用某个账户的该权限。

要创建新群组，请完成以下步骤。

(1) 从顶部菜单栏中，点击 Admin area（管理区域）的齿轮图标。
(2) 找到并点击 New group（新建群组）。此时将会显示群组创建表单（图 12-7）。
(3) 输入每个表单字段的详细信息。

- Group path（群组路径）
  该群组使用的 URL 片段。你只能输入小写字母和连字符。在 URL 中，它的用法与用户名相同。
- Details（详细信息）
  你的团队、组织、软件项目的简单描述，也就是“关于”或“生平”字段。
- Group avatar（群组头像）
  群组显示的图标。

(4) 找到并点击 Create group（创建群组）。此时将会显示项目的管理页面。

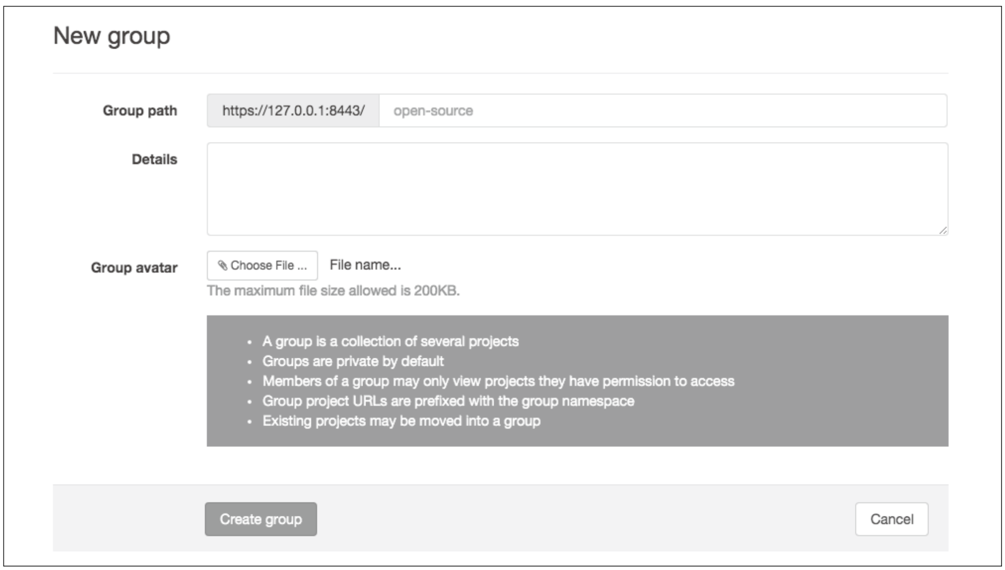


图 12-7：创建一个新群组

你的新群组已经创建。你现在可以为你的群组添加成员和项目。

12.4.1 添加群组成员

权限主要在项目中设置，而不是在群组中。然而，如果用户被授权担任团队中的特殊角色，那么他们能够执行下面一些其他操作。

- 所有人都能浏览群组。
- 只有 Owner（所有者）可以编辑群组，管理群组成员和删除群组。
- 群组的 Master 也可以在群组中创建项目。

要在群组中添加成员并为他分配角色，请完成以下步骤。

(1) 在顶部菜单中，点击 Admin area（管理区域）的齿轮图标。
(2) 在左侧边栏中，点击 Groups 菜单链接。此时将会显示群组管理页面。
(3) 找到表单来向群组添加用户（图 12-8）。

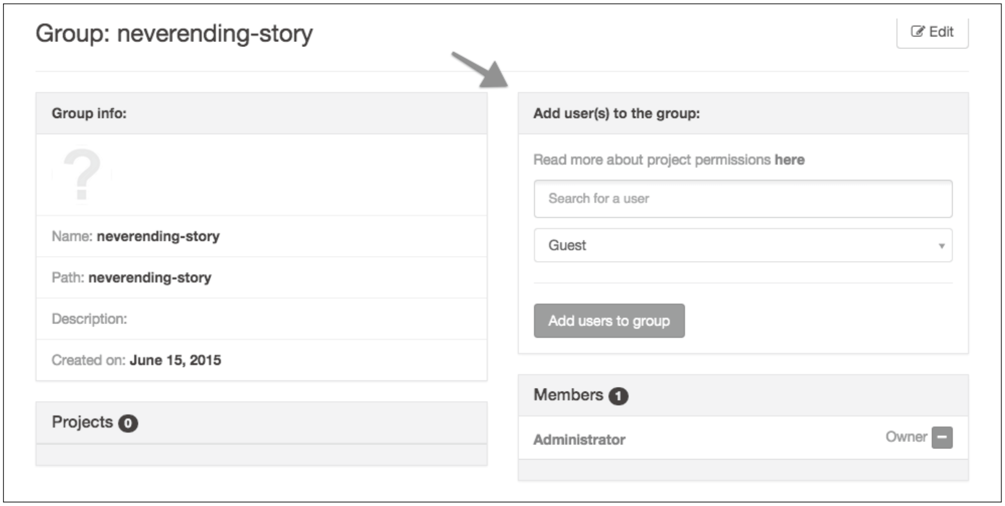


图 12-8：将用户添加到群组

(4) 为每个你想要添加到群组的用户输入以下详细信息。

- Username（用户名）
  你可以添加多个具有同一角色的用户（图 12-9）。

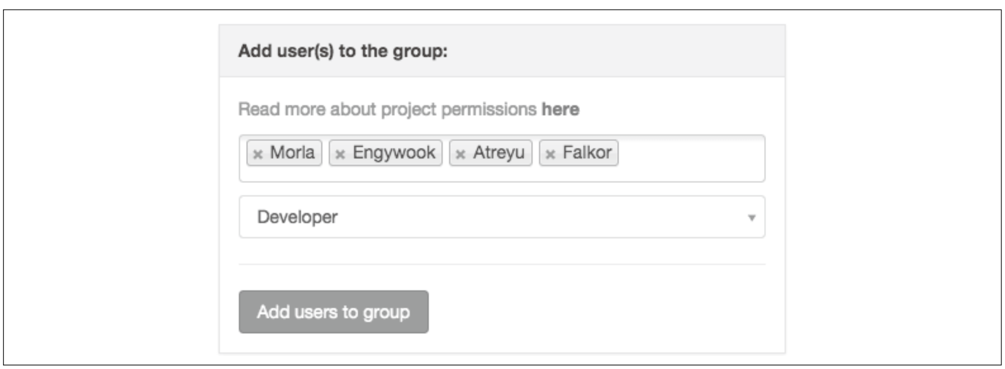


图 12-9：你可以在群组中同时添加多个用户，只要他们的角色相同即可

- Role（角色）
  选择 Guest（访客）、Reporter（报告者）、Developer（开发者）、Master 或 Owner（所有者）中的一个。

(5) 点击 Add users to group（将用户添加到群组）。

只有至少将一个项目添加到群组中，群组才可见。

12.4.2　将项目添加到群组

将项目添加到群组很简单，只是将命名空间调整为群组，而不是单个账户。

要在群组中创建新项目，请完成以下步骤。

(1) 在顶部菜单中，点击标签为 New project（新项目）的 + 图标。
(2) 输入 Project path（项目路径），仅可使用小写字母和连字符。
(3) 在 Namespace（命名空间）标签的旁边，点击下拉箭头并选择合适的群组（图 12-10）。
(4) 与你之前创建新项目时一样，完成每个字段的填写。
(5) 点击 Create project（创建项目）。



图 12-10：Sea of Possibilities 项目被添加到 Neverending Story 群组

新项目已经创建，并可用于开发。

如果项目已经存在，而你希望将它移到一个不同的命名空间（不同账户或群组），请完成以下步骤。

(1) 在顶部菜单中，点击 Admin area（管理区域）的齿轮图标。
(2) 在左侧边栏中，点击 Projects（项目）。
(3) 找到你想要重新分配的项目并点击它的名称。此时将会显示这个项目的管理摘要。
(4) 找到项目转移表单（图 12-11）。

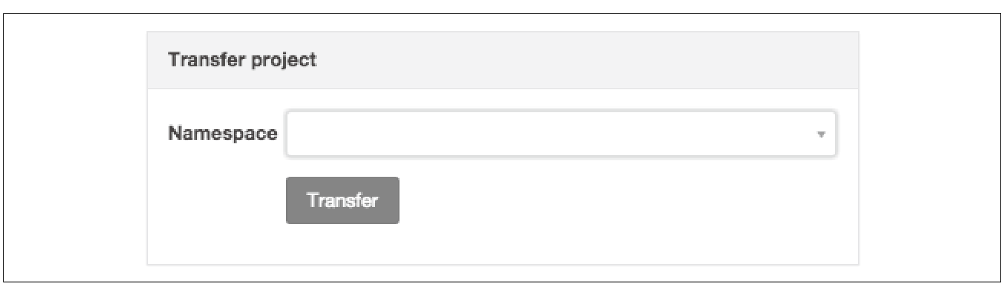


图 12-11：项目转移表单允许你将项目移动到一个不同的命名空间

(5) 在项目转移表单中，点击下拉框中的下箭头。此时将会显示群组和用户列表。选择你想要将项目转移到的群组。

(6) 点击 Transfer（转移）。

项目已经被转移到新的群组。之前的群组成员将不再拥有项目的权限。拥有项目本地克隆的用户需要更新项目的新命名空间的 URL。（阅读第 5 章来了解使用远程的详细信息）。

12.5 访问控制

项目可见性设置和每个账户角色的设置都可以限制项目访问。有了这两个选项，你可以灵活地掌握项目是如何管理的。在第 2 章中，你学到了连接仓库的许多不同方式，让人们拥有正确的访问等级。使用 GitLab 细致的管理，你可以确保每个人的实际权限与你希望他们得到的权限相同。

12.5.1 项目可见性

在给定项目中，你可以控制每个项目和每个角色的访问等级。

- Private（私有）
  为每个用户明确授权项目访问
- Internal（内部）
  项目可以被任何登录用户克隆
- Public（公开）
  项目不需任何验证即可被克隆

要调整项目的可见性设置，请完成以下步骤。

(1) 在顶部菜单中，点击 Admin area（管理区域）的齿轮图标。
(2) 在左侧边栏中，点击 Projects（项目）。
(3) 找到你想要调整的项目并点击它的名称。
(4) 在项目管理摘要页面中，找到并点击 edit（编辑）按钮。
(5) 找到表单中的 Visibility Level（可见性）部分（图 12-12）并调整你希望人们拥有的访问等级（私有、内部或公开）。
(6) 找到并点击 Save changes（保存更改）。



图 12-12：将项目可见性更新为 Public（公开）、Internal（内部）或 Private（私有）中的一个

你的项目可见性设置已经调整完毕。

12.5.2 使用项目角色限制活动

一旦用户能够看到一个项目，你就可以通过为每人分配一个特定的角色，进一步控制他们能够在仓库中进行的活动。你可以在你安装的 GitLab 中从目录 help/permissions/permissions 看到所有权限的综合对照表。

每个角色可用的功能快速摘要如下所示。

- Guest（访客）
  能够创建 issue 并留下评论，仅此而已！对于不需要代码权限，但需要参与到项目开发过程中的利益干系人来说很适合。
- Reporter（报告者）
  除了访客权限之外，报告者能够克隆仓库并创建代码片段。你可能希望给 CTO 这个角色，因为他们应该不再写代码了。（我只是在开玩笑。我觉得如果经理能够介入帮忙是最好的。我还认为经理应该专注于只有他们能够完成的外向任务。）
- Developer（开发者）
  除了之前的所有权限之外，开发者还可以创建新分支、创建合并请求、推送到不受保护的分支、添加标签、编写 Wiki 页面、管理 issue 跟踪工具，等等。团队中大多数人将会被分配该角色。你仍然可以限制他们对每个分支的访问，因此现在权限宽泛一些也没问题。
- Master
  除了之前的权限，Master 还可以创建里程碑、添加新的团队成员、推送到受保护分支、添加产品密钥，并且编辑项目本身。这个角色适合团队负责人以及想要经常改变团队组成或访问的项目经理。
- Owner（所有者）
  最后的角色同样可以改变项目可见性，将项目转移到另一个命名空间，以及删除项目。团队外的管理员适合拥有该角色。

要在团队中更新某个人的角色，请完成以下步骤。

(1) 在顶部菜单中，点击 Admin area（管理区域）的齿轮图标。
(2) 在左侧边栏中，点击 Projects（项目）。
(3) 找到你想要更新的项目。在项目的名称旁边有一个标签为 edit（编辑）的按钮。点击这个按钮。你将从管理区域重定向到该项目。
(4) 在左侧边栏中，点击 Members（成员）。
(5) 找到并点击 Edit group members（编辑群组成员）。成员列表将会被转换成一个配置列表。
(6) 找到你想要更改角色的用户，点击铅笔图标。一个新的下拉框将会出现（图 12-13）。
(7) 更新下拉框，使它包含该用户的合适角色。
(8) 点击 Save（保存）。



图 12-13：更新指定团队成员的角色

新的角色已被应用。

12.5.3　使用保护分支限制访问

GitLab 提供的最终访问层级是每个分支的访问设置。默认情况下，*master* 分支是受保护的，拥有 Developer（开发者）角色的用户无法推送到这个分支。他们需要使用合并请求流程来将他们的工作并入仓库中的 *master* 分支。如果你倾向于使用共享访问模型，则可以移除这个保护。

为了更新指定项目中受保护的分支，这个分支必须已经存在。一旦它存在于仓库中，你就可以打开或关闭访问。（记住，当你刚创建项目时，为新分支选择的是*默认访问*。）

要设置指定分支的访问控制，请完成以下步骤。

(1) 在顶部菜单中，点击 Admin area（管理区域）的齿轮图标。
(2) 在左侧边栏中，点击 Projects（项目）。
(3) 找到你调整的项目，点击名称旁边标签为 edit（编辑）的按钮。此时将会显示项目页面，而不是管理页面。
(4) 在左侧边栏中，依次点击 Settings（设置）和 Protected branches（分支保护）。
(5) 在下拉菜单中，选择你想要保护的分支（图 12-14）。
(6) 找到并点击 Protect（保护）按钮。

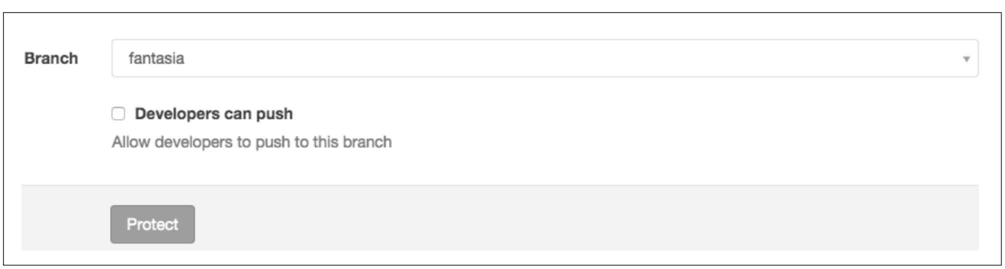


图 12-14：锁定分支，以便它仅可以接收来自拥有此项目或团队的 Master 或 Owner（所有者）角色的账户的更新

Developer（开发者）角色将不再能够更新这个分支。

要移除限制，请在同一个页面中完成以下步骤。

(1) 找到标题为 Already Protected（已经受保护）的部分（图 12-15）。
(2) 找到你想要更新的分支。
(3) 点击 Unprotect（取消保护）按钮。



图 12-15：已经受保护的分支可以被取消保护

现在，拥有 Developer（开发者）角色的用户将能够向你刚更新的分支推送提交。

12.6 里程碑

在每个项目中，你可以创建里程碑。它们被用于收集 issue、参与者和截止日期。如果你使用 Scrum 风格的敏捷开发，就会发现它们有助于冲刺的启动。一个群组共享的项目里程碑可以在一个报表页面中查看。这使得项目间的协调更加容易。但是，它仍然是每个仓库自身的，因此无法被当作强大的项目管理工具来允许你将不同代码库中的 issue 收集到一个项目中以方便管理。如果你在群组的所有项目中使用同一个名称，那么也可以偷个懒，将相关的事项放在一起。

要为你的项目创建一个新的里程碑，请完成以下步骤。

(1) 前往项目页面。
(2) 在左侧边栏中，点击 Milestones（里程碑）。
(3) 找到并点击 New milestone（新建里程碑）按钮。
(4) 为你的新里程碑完成以下表单字段（图 12-16）。

- Title（标题）
- Description（描述），附有其他文件
- Date（日期）

(5) 找到并点击 Create milestone（创建里程碑）按钮。

你的新里程碑已经成功创建。

要查看你的一个群组的所有里程碑列表，请完成以下步骤。

(1) 在页面右上角，点击你的用户头像。
(2) 在左侧边栏中，点击 Groups（群组）。
(3) 在群组列表中，点击你想要查看里程碑的群组名称。
(4) 在左侧边栏中点击 Milestones（里程碑）。

此时将会显示该群组所有里程碑的列表（图 12-17）。

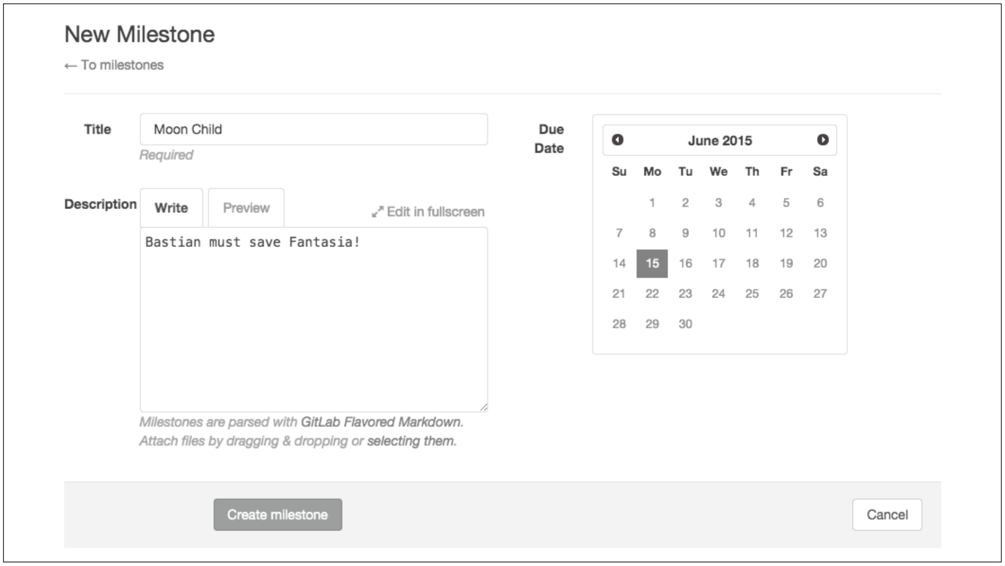


图 12-16：你可以为你的项目创建基于日期的里程碑

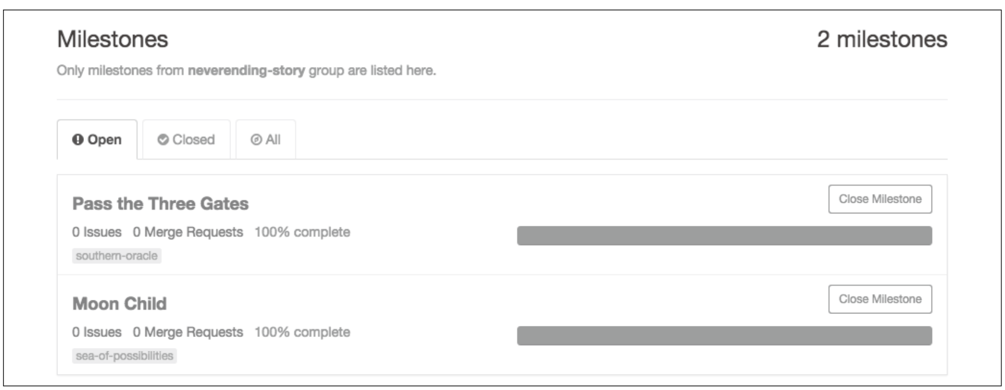


图 12-17：指定群组中所有里程碑的列表

12.7　小结

GitLab 是一个强大的开源代码托管系统，该系统的功能完全可以与 GitHub 和 Bitbucket 提供的功能相媲美。你可以在你自己的网络中安装 GitLab。

- 可以为每个仓库（可见性设置）、每个账户（角色设置）和每个分支（分支保护）自定义访问控制。
- 你可以将项目（仓库）和用户收集到群组，实现更方便的管理。
- 如果你不希望自己维护软件，GitLab 同样在 GitLab.com 提供了一个免费的云端托管服务。

附录A

奶油塔

在 Git 中，分支用于维护代码中的变化。这些变化可能是正在进行的工作，也可能是一个完全不同的方向。这些分支看上去很像家庭食谱的不同变种。本附录包括我家的一款经典加拿大甜点奶油塔（https://en.wikipedia.org/wiki/Butter_tart）的两个食谱变种。（加拿大以外的人看到这里，会对这个甜点的馅料产生争议。这有点像变基，但是更糟。）

A.1 Austin的奶油塔

这是我母亲的食谱，是从她的祖母 Granny Austin 那里传下来的。该食谱中一向只添加黑加仑，而不添加任何其他馅料。

面皮

- 两杯半面粉
- 一杯起酥油
- 一撮盐
- 冰水（足以混合）

(1) 将起酥油切碎，加入面粉中。
(2) 添加冰水（大约半杯）。
(3) 用叉子搅拌。
(4) 铺开。
(5) 在盛松饼的托儿中戳几个洞，不要填满，450°F 烘焙 12 分钟。

内馅

- 一杯糖

- 半杯软化黄油
- 三个鸡蛋
- 两汤匙甜奶油或酸奶油

(1) 混合内馅材料。
(2) 在填充了面皮的松饼托上，400°F 烘焙 25 分钟。

A.2 van der Heyden的奶油塔

这是我伯母的食谱，是从她的母亲 Pat van der Heyden 那里传下来的。通常不需要另外添加什么，但可以加一些烘烤的碎果仁、巧克力脆片或葡萄干。

内馅

- 三分之二杯软化黄油
- 三杯红糖
- 三杯玉米糖浆
- 十二个鸡蛋

混合黄油和糖。先添加玉米糖浆，然后添加鸡蛋。搅拌。使用你最喜欢的面皮配方，铺平后切成蛋挞合适的大小，就像一个松饼托的形状。用叉子在每个面皮底部戳几个洞。塞入奶油塔的内馅。在 400°F 下烘焙 21 分钟（左右）。

可选择性添加的食材包括以下这些。

- 烘烤的碎果仁
- 巧克力脆片
- 葡萄干

面皮

- 六杯普通面粉
- 三杯起酥油（Karin 会用 Crisco 起酥油，而她母亲则用猪油）
- 两个鸡蛋
- 一点醋加上两杯冰水

用起酥油混合面粉，使之结块。打蛋，加醋和水。不断向干燥的面粉加水直到稳定。把没用完的面皮装在塑料盒子里冷冻供下次使用。

附录B

安装最新版本的Git

本书主要介绍 Git 的基础知识，如果不升级 Git，你会遗漏很多新的功能。一般来说，我认为新版本的软件更易于使用。错误说明更加清晰，并提供了更好的“下一步操作”建议。一些复杂的命令格式得以改进，使得命令更加易于使用。（例如，使用 `--delete` 参数删除远程分支的能力，而不是使用冒号这种奇怪的格式。）

你觉得自己已经安装了 Git。很好！

但你的操作系统自带的版本十有八九是过时的。“对我来说都一样”，我听到你这么说了。我知道，我知道。我曾经也是这么想的：Git 那么陈旧复杂，几百万年没变过了吧，直到我去参加一个 Git 开发者会议。在会议中，我遇到了很多热情友好、耐心有趣并积极改进 Git 的开发者。在那时，Git 的维护者是 Junio Hamano，而 Windows 的维护者是 Johannes Schindelin。他们都在会议现场，并且真心希望让 Git 更易于用户使用。如果你不安装最新版本，就看不到社区快速的发展。

你应该总是试着使用最新版本的软件，而且必须确认你至少在使用 2.5 以上版本的 Git。在这个版本的 Git 中，`git help` 命令好用多了。我对这个改进非常激动，因为这是我初识 Git 时困扰我很久的事情。然后，在 Git 开发者大会上，我不经意间的评论最后变成了一个非正式的 bug 报告……几个月后 Sébastien Guimmara 和 Eric Sunshine 将我的愿望变成了你们可以使用的命令。这真是难以置信！

我通常只会落后几个补丁版本（例如，如果最新版本是 2.5.2，我或许用的是 2.5.0），但我确实会留意保持相对最新。如果你不记得最近几个月安装过 Git，那么几乎肯定需要升级了。如果你的系统中还没有 Git，那么你还需要安装它（这很简单！你可以使用安装程序！）。

B.1 安装和升级Git

在 Windows 和 OS X 下都有人性化的 Git 安装程序。一般来说，安装程序会在你升级 Git 时试图保持你的设置不变。

安装程序可以从下面的网页下载：

http://git-scm.com/downloads

如果你使用 Linux 或 Unix 系统，那么可能已经安装了 Git，但应该升级到最新版本。使用你的包管理工具来完成这一步（见 B.3 节）。OS X 用户或许同样希望使用包管理工具来安装并更新 Git。

B.2 查找命令行

这本书的重点是介绍从命令行使用 Git。对此我一点都不后悔。出于以下两个关键的原因，我认为你应该尝试一下从命令行使用 Git。

(1) 当所有人都在命令行工作时，复制粘贴所有操作系统中可用的文档更加容易。
(2) 当你在命令行工作时，会获得更好的错误消息。在图形化界面上，你很难复制粘贴你在遇到麻烦之前运行的一系列命令。通过在命令行中工作，你将能够在出错时更快地获得别人的帮助。

如果你喜欢图形化界面，那么随着你渐渐熟悉本书中的概念，我建议你将知识迁移到图形化界面上。

B.2.1 OS X

(1) 打开 Spotlight。Spotlight 可以在菜单栏右上角的放大镜或按下 Control+ 空格找到。
(2) 在 Spotlight 的搜索框中键入 terminal，然后按下回车。一个新的命令行窗口将会出现。

B.2.2 Linux

命令行窗口的位置将会根据你使用的 Linux 发行版和你使用的窗口管理器而有所不同，如果你不知道如何在你的 Linux 版本中打开命令行窗口，使用你熟悉的搜索引擎应该就能很快弄明白。

B.2.3 Windows

你使用的方法将会根据正在使用的 Windows 版本而略有不同。

针对 Windows 7 的方法如下所示。

(1) 点击“开始”按钮。
(2) 选择“应用程序”→“附件”→“命令提示符”。命令行窗口将会打开。

针对 Windows 8 的方法如下所示。

(1) 前往“应用”页面（手指上滑，或使用鼠标点击屏幕下方的下箭头）。
(2) 通过滑动或向右滚动，找到标题为“Windows 系统”的分组。
(3) 在“Windows 系统”分组下，按下或点击“命令提示符”。

B.3 在*nix系统上升级

包管理工具是确保你在系统中使用最新版本的 Git 的绝佳方式。在 Linux 和 Unix 变种上，你将会使用之前安装 Git 时用过的同一个包管理工具来升级 Git（好吧，Git 之前已经安装了，你可能需要升级一下）。

Homebrew 是 OS X 下的一个包管理工具

如果你正在使用 OS X，并且已经安装了 Homebrew（http://brew.sh/），你应该使用这个包管理工具来更新 Git。

在使用包管理工具时，你需要记得保持你的包列表最新。一般来说你可以通过包管理工具的子命令 `update` 来完成更新。例如，在 Ubuntu 上我会使用 `apt-get update`，在 Fedora 上我会使用 `yum check-update`，而在 OS X 上我会使用 `brew update`。

一旦包列表更新后，你可以为你的系统安装最新包版本的软件。这通常是通过子命令 `install` 或 `upgrade` 来完成的。

针对 OS X 的安装命令如下所示。

```
$ brew install git
```

针对 Ubuntu 和使用 `apt` 的 Linux 发行版的安装命令如下所示。

```
$ apt-get install git
```

针对 Fedora 和使用 `yum` 的 Linux 发行版的安装命令如下所示。

```
$ yum install git
```

为了确保你的包是最新的，你可以单独或按需更新它们（例 B-1）。这通常通过子命令 `upgrade` 来完成，尽管如果更新的包可用，再次运行 `install` 命令也能够更新软件。

谨慎升级

小心！包管理工具绝大多数时候都是很棒的，但有时候在你的截止期限前更新所有包并非明智的选择。

例 B-1　使用 Brew 升级包

OS X 下只升级 Git，如下所示。

```
$ brew upgrade git
```

OS X 下通过 Homebrew 升级所有包，如下所示。

```
$ brew upgrade
```

B.4　OS X上的小麻烦

当我与 Git 社区更加熟悉的时候，我开始使用自定义的构建而不是安装程序，这样我可以尝试最新的功能和更新的文档。当我尝试将代码推送到远程仓库时，有时会遇到下面的错误。

```
git: 'credential-osxkeychain' is not a git command. See 'git --help'.
```

出于一些原因，我的环境变量与我的预想不符。在厌倦了一遍遍的尝试之后，我下载了另一份钥匙串辅助程序，将它放在我的硬盘上的已知位置。

你不太可能会丢失你的钥匙串

我高度怀疑你需要这一节，这对我来说是一个为未来准备的小贴士，记录我之前是如何解决这个问题的。（是的，我把自己的书当作参考。我写下重要的东西，这样就不需要将他们都记在我的脑子里。）

首先，验证你在全局的 Git 配置文件中设置好了正确的验证工具。这个文件位于目录 ~/.gitconfig 下，应该包括如下设置。

```
[credential]
        helper = osxkeychain
        useHttpPath = true
```

如果你在配置文件中看不到这些，现在运行以下命令进行设置。

```
$ git config --global credential.helper osxkeychain
```

运行以下命令来查看问题是否已解决。

```
$ git credential-osxkeychain
```

你应该不会收到之前看到的错误消息。

如果你还是再次看到了错误消息，遵循以下指令。你将下载并安装另一份 `osxkeychain` 的辅助程序，如下所示。

```
$ curl -s -O http://github-media-downloads.s3.amazonaws.com/
        osx/git-credential-osxkeychain
```

调整文件权限来运行这个程序，如下所示。

```
$ chmod u+x git-credential-osxkeychain
```

将这个辅助程序移动到 Unix 程序的应用文件夹下，这个程序会以根用户身份运行，所以你需要输入你的 OS X 登录密码来运行这个命令，如下所示。

```
$ sudo mv git-credential-osxkeychain /usr/local/git/bin
```

现在，当你运行下面的命令时，应该不会看到之前出现的关于命令找不到的错误。

```
$ git credential-osxkeychain
```

这份文档是从“Mac OS X 上的 Git 和 GitHub 新手设置指南”（http://burnedpixel.com/blog/setting-up-git-and-github-on-your-mac/）中的说明转换而来。Chris Chernoff，如果你看到了这里，谢谢你！你的建议帮助我节省了运行自定义 Git 构建的时间，想要将本书更新的分支推送到 Atlas 构建服务器上时，不用每次都输入我设置的 42 位字符的随机密码。

B.5 在命令行上访问Git帮助

Git 提供了内置的命令行文档。这些信息可以通过运行以下命令获得。

```
$ git help
```

你可以通过指定话题名称阅读某个话题的所有文档，如下所示。

```
$ git help topic
```

为了前往帮助页面，你可以使用键盘的方向键来上下滚动，当你结束文档的阅读时，按 `q` 退出。

要查看所有话题的列表，请使用以下命令。

```
$ git help --all
```

你还可以查看 Git 的术语词汇表，如下所示。

```
$ git help glossary
```

附录C

配置Git

随着时间的增长，你会发现在命令行中帮助你使用 Git 的捷径。我自己发现，那些对 Git 最感到沮丧的人往往是最少定制 Git 的人。在使用 Git 时，你将会设置两种类型的配置项：全局设置，应用于所有仓库；本地设置，只应用于当前仓库。全局设置的例子可以是你的名字，而你的电子邮件可能根据你的个人项目和工作项目配置而不同。

全局设置储存在 ~/.gitconfig 文件中，而本地设置储存在你正在工作的仓库中的 .git/config 文件中。如果需要，你始终可以回去编辑你的设置。

你可以查看当前设置的值。例如，例 C-1 向你展示如何查看我的名字。

例 C-1　显示一个设置的值

```
$ git config --get user.name
```

你还可以获得当前设置过的所有值的列表（例 C-2）。

例 C-2　显示所有当前设置过的所有配置项的值

```
$ git config --list
```

所有变量的列表可以从 config 的命令页面（http://git-scm.com/docs/git-config）上查询到。这也同样可以通过运行下面这个命令来访问。

```
$ git help config
```

C.1　表明身份

为了展现自己的工作，你需要告诉 Git 你是谁。我们会在全局储存你的名字（例 C-3）和电子邮件（例 C-4）。因为这是一个全局设置，所以你不需要在某个特定的仓库中进行更改。

例 C-3　配置你的名字

```
$ git config --global user.name 'Your Name'
```

例 C-4　配置你的电子邮件地址

```
$ git config --global user.email 'me@example.com'
```

在某些仓库中使用特定的电子邮件地址是适当的（例如，如果你进行的是个人项目）。你可以通过完成以下步骤，指明应该仅将这些更改应用于特定仓库。

(1) 前往拥有想要配置的仓库的目录。
(2) 用 `--local` 替换 `--global`，然后应用配置命令。

示例如下所示。

```
$ git config --local user.email 'me@work.com'
```

C.2　更改提交说明编辑器

默认情况下，Git 会使用系统编辑器。在 OS X 和 Linux 上，一般这是 Vim。我非常喜欢 Vim 而且这也是我使用的编辑器。它有一些难以驾驭，因此你可能会希望更改你的编辑器。

通过运行以下命令查看 Git 当前使用的编辑器。

```
$ git config --get core.editor
```

你必须先退出再提交

在你退出编辑器后提交才会储存到 Git，而不只是保存提交说明。这可能会影响你对文本编辑器的选择。

如果你要使用 Textmate，请使用以下命令。

```
$ git config --global core.editor mate -w
```

如果你要使用 Subline，请使用以下命令。

```
$ git config --global core.editor subl -n -w
```

如果你想要更改 Windows 的编辑器，则需要加入应用文件的完整路径。应用通常安装在目录 C:\Program Files 下，你需要在路径两侧加上双引号。此外，每当你使用 Bash 来调用 `git config` 时，必须给值加上引号，这会导致出现了一个双重引号的字符串，如下所示。

```
$ git config --global core.editor '"C:\Program Files\Vim\gvim.exe" --nofork'
```

至于其他编辑器，请查看你选择的编辑器的配置说明。

C.3　添加颜色

阅读大段文本是很费劲的。可以在命令行中添加辅助颜色，以便更容易看清楚 Git 在做什么，如下所示。

```
$ git config --global color.ui true
$ git config --global diff.ui auto
```

C.4 自定义命令提示符

如果你在命令行中工作，那么在明确询问 Git 之前，你对文件中发生了什么是毫无线索的。不断地询问是非常令人厌烦的。就好像你在八岁的时候坐在车后座，不断地向司机发牢骚“我们快到了吗”。

代替明确询问的是，我修改了命令提示符，让它告诉我当前签出的分支，以及我是否修改了仓库中的任何文件，这是一个很常见的小技巧，但每个开发者都有自己的实现方式。上网搜索“bash prompt git status”将会返回大量结果。我自己的提示符非常简单，但其他人在他们的提示符中添加了很多细节。例如：在命令提示符中显示（带颜色的）Git 状态和分支（https://git-scm.com/docs/git-config）或本地文件状态（https://github.com/magicmonty/bash-git-prompt）。与所有技术一样，你一开始添加的东西越多，那么它出错后你需要解决的 bug 就越多。

我发现这些华丽的提示符很难设置，最终变成一个非常复杂的提示符。我推荐从简单的开始，然后如果你真的想要更多信息，再往上添加。颜色上的简单变化以及分支的名称，事实上对我来说已经足够了，去掉所有额外信息会显得更加清爽。

C.5 忽略系统文件

我们都曾遇到过：不小心地暂存了 OS X 的 .DS_Store 系统文件或者文本编辑器的临时 .swp 文件。你可以通过设置一个全局忽略文件来减少尴尬，让 Git 避免将这些文件提交到任何你创建的本地仓库。可以找到忽略文件的详细清单（https://github.com/github/gitignore）。挑选最适合你的系统和项目的列表。

一旦你选好了想要忽略的文件列表，请完成以下步骤。

(1) 创建一个新的文本文件，名为 .gitignore_global，将它放在你的 home 目录下。
(2) 通过运行以下命令告诉 Git 使用这个配置文件。

```
$ git config --global core.excludesfile ~/.gitignore_global
```

你或许还有特定项目中的文件或者甚至是输出目录（例如构建文件夹）不想提交到仓库。在每个仓库中，你都可以创建一个自定义的“忽略”文件，进一步限制 Git 要跟踪的文件，步骤如下。

(1) 创建一个新的文本文件，名为 .gitignore，将它放在你的项目根目录下。
(2) 在这个文件中添加所有你希望 Git 永远不要添加到仓库的文件名称，每个文件名应该单独一行。你也可以使用模式匹配，例如 `*.swp` 代表临时的编辑器文件。

这个更改需要在 Git 仓库中创建一个新提交，如下所示。

```
$ git add .gitignore
$ git commit -m "Adding list of files to be ignored."
```

C.6　行结束符

如果你在跨平台的团队中工作，开发者分别使用 OS X、Linux 和 Windows，那么本节尤其重要。

你应该在全局设置这个行结束符，但同时在每个仓库中添加这个设置，从而确保没有显式设置行结束符的开发者不会遇到问题，如下所示。

```
$ git config --global core.autocrlf input
```

为了明确地让所有贡献者使用正确的行结束符，你需要在你的仓库中添加一个 .gitattributes 文件，标记正确的行结束符、应该被改正的文本文件和不应该被修改的二进制文件。

在仓库根目录（与 .git 位于同一文件夹中）下创建一个名为 .gitattributes 的新文本文件。新文件的样例如下所示。

```
# 设置所有文件的默认行为
* text=auto

# 列出应使用系统相关的行结束符的文本文件
*.php text
*.html text
*.css text

# 列出应使用CRLF行结束符且不根据本地操作系统转换的文件
*.sln text eol=crlf

# 列出所有不应进行修改的二进制文件
*.png binary
*.jpg binary
*.gif binary
*.ico binary
```

将文件添加到暂存区索引，如下所示。

```
$ git add .gitattributes
```

将文件提交到仓库，如下所示。

```
$ git commit -m "Require the right line endings for everyone, forever."
```

修复行结束符

如果你不幸在项目中期需要统一行结束符，则需要完成以下步骤。

(1) 与你的团队一起决定你的仓库中使用的“官方”行结束符。
(2) 变基每个受影响的文件来重置行结束符。如果是我的朋友，她会使用 Vim 并设置 `:set ff=unix`。你可能更喜欢在你的文本编辑器中打开每个文件并重新保存每个文件来重置行结束符，或者使用 `dos2unix` 这样的命令行工具。
(3) 将更新的文件添加并提交到仓库。
(4) 如上节所述的那样将 .gitattributes 文件添加到你的仓库。

(5) 将更改推送到代码托管服务器。
(6) 要求团队中其他人使用 `rebase` 命令更新他们的工作，这样“不好的”行结束符不会再被意外地引入仓库了。
(7) 喝一杯热巧克力或威士忌。这是你应得的。

附录D

SSH密钥

使用了 SSH 密钥之后，在你连接到远程机器时就不需要每次输入密码。密钥本身是成对出现的：一个公钥和一个私钥。应该将私钥当作密码对待，且永远不应该与别人共享。公钥将会被“安装”在其他地方，例如代码托管系统中。

D.1　创建你自己的SSH密钥

要想创建一个 SSH 密钥，你需要运行一个程序，它会保存一对文件。这个必需的软件已安装在 *nix 系统上，而 Windows 用户需要额外（免费）下载这个软件。

D.1.1　Linux、OS X和Unix变种

为了生成一对密钥，你需要运行以下命令。

```
$ ssh-keygen -t rsa -b 4096 -C "your_email@example.com"
```

系统将提示你输入以下信息。

- 文件位置
  通过按下回车接受默认位置以继续。
- 密码
  可选项。但你真的应该设置一个。使用容易记忆的密码或者将它放在你日常使用的非常安全的密码管理器中。

密钥的指纹将会输出到屏幕上，且这个密钥对将会被保存到合适的位置 ~/.ssh/。

现在，你需要在系统中注册这个密钥，才能开始使用它。

此时事情变得有些神秘。你需要在本地“代理”中注册你的密钥（在用 OS X？想象一下“钥匙串”，但不一样）。启动 SSH 代理应用并重定向使用 Bourne，如下所示。

```
$ eval "$(ssh-agent -s)"
```

使用代理注册 SSH 密钥，如下所示。

```
$ ssh-add ~/.ssh/id_rsa
```

你的密钥已经注册。

如果你需要立即使用这个密钥，请直接跳到 D.2 节。

D.1.2　Windows

在 Windows 下生成一个 SSH 密钥对（需要用到 PuTTYgen 软件）的操作步骤如下所示。

(1) 在 PuTTYgen 下载页面（http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html）上找到 PuTTYgen 的最新版本。文件名为 puttygen.exe。
(2) 右键点击 puttygen.exe 的链接，选择“将链接另存为”。具体文字可能根据你的浏览器而略有不同。
(3) 出现弹出窗口后，选择一个你能够轻易找到的目录（例如，你的桌面目录）。
(4) 找到桌面上的 PuTTYgen 应用。双击图标运行这个程序。
(5) 在窗体底部的 Type of key to generate（生成的密钥类型）中选择 SSH-2 RSA。
(6) 找到并点击 Generate（生成）按钮。
(7) 摇摆你的鼠标。我是认真的。你将会制造出随机数据（噪声），这有助于密钥生成的过程。继续操作，直到进度条全部充满。
(8) 系统将提示你输入密码。这是可选的，但你应该添加一个。
(9) 找到并点击 Save private key（保存私钥）。
(10) 找到并点击 Save public key（保存公钥）。

这样应该会将密钥保存到目录 ~/.ssh/ 下的某个合适位置。

如果你准备好立即使用 SSH 密钥，请同样完成以下几步。

(1) 找到标题 Public key for pasting into OpenSSH authorized_keys file（粘贴到 OpenSSH 的 authorized_keys 文件中的公钥）。
(2) 右键点击标题下方的随机字符串。
(3) 选择 Select all（全选），然后选择 Copy（复制）。

你的公钥已经被复制到剪贴板上。你可以进行下一步。

D.2　获得你的SSH公钥

当你的代码托管系统向你询问“SSH 公钥”时，它需要 id_rsa.pub 文件中的内容。这个文件通常储存在 home 目录下的一个隐藏文件夹中：.ssh。要找到这个文件，并将它的内容复制到剪贴板，根据你的操作系统，你需要完成下述命令。通过在命令行中工作，你可以避

免寻找一个能识别出 .pub 文件的编辑器。它只是文本，但你安装的文本编辑器可能并不知道这一点。

在 OS X 下的操作步骤如下所示。

(1) 打开命令行窗口。
(2) 运行下列指令：`cat ~/.ssh/id_rsa.pub | pbcopy`。

在 Linux 下的操作步骤如下所示。

(1) 打开命令行窗口。
(2) 运行下列指令：`cat ~/.ssh/id_rsa.pub`。此时应该会在屏幕上输出一个非常长的字符串，它应该与命令行窗口等宽，并且不包含单词 PRIVATE KEY。如果找不到文件，你需要首先创建一个 SSH 密钥。
(3) 复制在屏幕上输出的所有文本。

在 Windows 下的操作步骤如下所示。

(1) 打开 Git Bash 窗口。
(2) 运行下列命令：`clip < ~/.ssh/id_rsa.pub`。它会将你的 SSH 公钥复制到剪贴板。

你的 SSH 公钥现在复制到剪贴板了，你可以将它粘贴到你选择的代码托管系统的设置页面中。

关于作者

Emma Jane Hogbin Westby 从 1996 年开始开发网站，最初作为一名开发者，现在是团队负责人。她从 2002 年开始教授 Web 相关技术，并且在全球举办过 100 多场会议演讲、课程和研讨会，内容涉及 Web 前端开发、无障碍标准、分布式版本控制、可视化和变更管理。她之前出版过两本 Web 开发相关的图书。

Emma 提倡通过动手创新性地参与技术，她还是一个业余的养蜂人。你可以在 Twitter 上关注她，她的 Twitter 账号是 @emmajanehw。

关于封面

本书封面上的动物是白鹡鸰（Motacilla alba）、灰鹡鸰（Motacilla cinerea）和黄鹡鸰（Motacilla flava）。

Motacilla 属的名字意为“移动尾巴”，正如它们的名字所暗示的，这些小巧而精力充沛的鸟因为经常上下扇动它们的长尾巴而被人所熟知。但这种行为的原因尚不明确。平均来说，这些鸟身长约 6 英寸，重约 0.8 盎司。

鹡鸰靠捕食小昆虫为生，偶尔在附近的牛群中觅食，捕捉干扰牛群的昆虫。它们也在地面上筑巢，一次下 4~7 个蛋。

鹡鸰分布广泛，繁殖于整个欧亚地区，有时还迁徙到非洲的热带地区。它们喜欢开阔的乡野，如农场和草地。然而，近年来这三个物种的数量都遭遇了严重的下滑，可能是由于农业的变化导致的。

O’Reilly 封面上的很多动物都已经濒危，每一种动物对我们的世界都很重要。要了解更多有关如何提供帮助的信息，请访问网页 http://www.oreilly.com/animals.csp。

该封面图片取材自 *Wood’s Illustrated Natural History*。

微信连接

回复“开发利器”查看相关书单

微博连接

关注@图灵教育 每日分享IT好书

QQ连接

图灵读者官方群I：218139230
图灵读者官方群II：164939616

图灵社区
iTuring.cn

在线出版，电子书，《码农》杂志，图灵访谈

O'REILLY®

Git团队协作

Git不仅是协助软件开发的利器，还是高效团队管理的秘密武器。曾在全球上百场会议中分享过Git精神的Emma，将在本书中与读者分享自己多年来在开发和项目管理中利用Git技能所得到的丰富经验。

书中内容共分为三部分。第一部分介绍工作流的构建，从宏观视角陈述以不同方式组织工作流会如何影响团队协作方式。第二部分分别针对单人团队和多人团队，从实践角度阐述Git命令，提供上手练习。第三部分介绍主流代码托管系统，为读者提供这些平台用法的入门指南。

- **探索团队构建的奥秘**
- **研究使用Git创造和部署软件的流程**
- **构建工作流来影响团队的协作方式**
- **了解实施代码评审的实用流程**
- **建立共享仓库，将特定的团队成员看作贡献者、消费者或维护者**
- **了解团队成员使用Git命令背后的原因**
- **使用分支策略来分隔项目中不同的工作**
- **了解三个主流协作平台的用法：GitHub、Bitbucket和GitLab**

Emma Jane Hogbin Westby，资深Web开发人员，并拥有丰富的团队管理经验，曾在全球举办过100多场会议演讲、课程和研讨会，内容涉及Web前端开发、无障碍标准、分布式版本控制、可视化和变更管理。Twitter账号：@emmajanehw。

"这本书视角独特，强调Git如何能够促进团队协作，让我收获颇丰……重点介绍了工作流和角色之间的沟通，引导读者理解实际项目中遇到的真实需求。无论是探索团队协作的不同方式，还是探索现代版本控制系统帮助推进项目的方法，都可以利用本书学会释放Git全部潜能的方法，让工作事半功倍。"

——Johannes Schindelin博士

Windows端Git维护者

PROGRAMMING

封面设计：Ellie Volckhausen 张健

图灵社区：iTuring.cn
热线：(010)51095186转600

分类建议 计算机／软件开发

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版

此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)

ISBN 978-7-115-45467-6

ISBN 978-7-115-45467-6
定价：69.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这可以找到我们：

微博 @图灵教育：好书、活动每日播报
微博 @图灵社区：电子书和好文章的消息
微博 @图灵新知：图灵教育的科普小组
微信 图灵访谈：ituring_interview，讲述码农精彩人生
微信 图灵教育：turingbooks